Optimizing VR Hit Game *Space Pirate Trainer* to Perform on Intel® Integrated Graphics

By Cristiano Ferreira (@cristianohh), Dirk Van Welden (@quarkcannon), and Seth Schneider (@SethPSchneider)

*Space Pirate Trainer* was one of the original launch titles for HTC Vive*, Oculus Touch*, and Windows* Mixed Reality. Version 1.0 launched in October 2017, and swiftly became a phenomenon, with a presence in VR arcades worldwide, and over 150,000 units sold according to RoadtoVR.com. It’s even the go-to choice of many hardware producers to demo the wonders of VR.

“I’m Never Going to Create a VR Game!”

These are the words Dirk Van Welden, CEO of I-Illusions and Creative Director of *Space Pirate Trainer*. He made the comment after first using the Oculus Rift* prototype. He received the unit as a benefit of being an original Kickstarter backer for the project, but was not impressed, experiencing such severe motion sickness that he was ready to give up on VR in general.

Luckily, positional tracking came along and he was ready to give VR another shot with a prototype of the HTC Vive. After one month of experimentation, the first prototype of *Space Pirate Trainer* was completed. Van Welden posted it to the SteamVR forums for feedback and found a growing audience
with every new build. The game caught the attention of Valve, and i-Illusions was invited to their SteamVR developer showcase to introduce Space Pirate Trainer to the press. Van Welden knew he really had something, so he accepted the invitation. The game became wildly popular, even in the pre-beta phase.

**What is Mainstream VR?**
Mainstream VR is the idea of lowering the barrier of entry to VR, enabling users to play some of the most popular VR games without heavily investing in hardware. For most high-end VR experiences, the minimum specifications on the graphics processing side require an NVIDIA GeForce* GTX 970 or greater. In addition to purchasing an expensive discrete video card, the player also needs to pay hundreds of dollars (USD) for the headset and sensor combo. The investment can quickly add up.

But what if VR games could be made to perform on systems with integrated graphics? This would mean that any on-the-go user with a top-rated Microsoft Surface* Pro, Intel® NUC, or Ultrabook™ device could play some of the top VR experiences. Pair this with a Windows Mixed Reality headset that doesn’t require external trackers, and you have a setup for VR anywhere you are. Sounds too good to be true? Van Welden and I thought so too, but what we found is that you can get a very good experience with minimal trade-offs.

![Figure 1: Space Pirate Trainer* - pre-beta (left) and version 1.0 (right).](image)

**Rendering Two Eyes at 60 fps on a 13 Watt SoC with Integrated Graphics? What?**
So, how do you get your VR game to run on integrated graphics? The initial port of Space Pirate Trainer ran at 12 fps without any changes from the NVIDIA GeForce GTX 970 targeted build. The required frame rate for the mainstream space is only 60 fps, but that left us with 48 fps to somehow get back through intense optimization. Luckily, we found a lot of low-hanging fruit—things that greatly affected performance, the loss of which brought little compromise in aesthetics. Here is a side-by-side comparison:
Getting Started

VR developers probably have little experience optimizing for integrated graphics, and there are a few very important things to be aware of. In desktop profiling and optimization, thermals (heat generation) are not generally an issue. You can typically consider both the CPU and GPU in isolation and can assume that each will run at their full clock-rate. Unfortunately, this isn’t the case for SoCs (System on a Chip). “Integrated graphics” means that the GPU is integrated onto the same chip as the CPU. Every time electricity travels through a circuit, some amount of heat is generated and radiates throughout the part. Since this is happening on both the CPU and GPU side, this can produce great amounts of heat to the total package. To make sure the chip doesn’t get damaged, clock rates for either the CPU or the GPU need to be throttled to allow for intermittent cooling.

For consistency, it’s very helpful to get a test system with predictable thermal patterns to use as a baseline. This enables you to always have a solid reference point to go back to and verify the performance improvements or regressions as you experiment. For this, we recommend using the **GIGABYTE ultra-compact PC GB-BKi5HT2-7200** as a baseline, as its thermals are very consistent. Once you’ve got your game at a consistent 60 fps on this machine, you can target individual original equipment manufacturer (OEM) machines and see how they work. Each laptop has their own cooling solution, so it helps to run the game on popular machines to make sure their cooling solutions can keep up.

![Figure 2: Comparison of Space Pirate Trainer at 12 fps (left) and 60 fps (right).](image-url)
The above system is currently used at Intel for all of our optimizations to achieve consistent results. For the purpose of this article and the data we provide, we’ll be looking at the Microsoft Surface Pro:

**Device specifications**

<table>
<thead>
<tr>
<th>Device name</th>
<th>GPA-MSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Intel(R) Core(TM) i7-7660U CPU @ 2.50GHz 2.50 GHz</td>
</tr>
<tr>
<td>Installed RAM</td>
<td>16.0 GB</td>
</tr>
<tr>
<td>Device ID</td>
<td>F8BBBBB15-B0B6-4B67-BA8D-58D5218516E2</td>
</tr>
<tr>
<td>Product ID</td>
<td>00330-62944-41755-AAOEM</td>
</tr>
<tr>
<td>System type</td>
<td>64-bit operating system, x64-based processor</td>
</tr>
<tr>
<td>Pen and touch</td>
<td>Pen and touch support with 10 touch points</td>
</tr>
</tbody>
</table>
For this intense optimization exercise, we used Intel® Graphics Performance Analyzers (Intel® GPA), a suite of graphics analysis tools. I won’t go into the specifics of each, but for the most part we are going to be utilizing the Graphics Frame Analyzer. Anyway, on to the optimizations!

**Optimizations**

To achieve 60 fps without compromising much in the area of aesthetics, we tried a number of experiments. The following list shows the biggest bang for the optimization buck as far as performance and minimizing art changes goes. Of course, every game is different, but these are a collection of great first steps for you to experiment with.

**Shaders—Floor**

The first optimization is perhaps the easiest and most effective change to make. The floor of your scenes can take up quite a bite of pixel coverage.

![Figure 5: Floor scene, with the floor highlighted.](image)

The above figure shows the floor highlighted in the frame buffer in magenta. In this image, the floor takes up over 60 percent of the scene as far as pixel coverage goes. This means that material optimizations affecting the floor can have huge implications on keeping below frame-budget. *Space Pirate Trainer* was using the standard Unity* shader with reflection probes to get real-time reflections on the surface. Reflections are an awesome feature to have, but a bit too expensive to calculate and sample every frame on our target system. We replaced the standard shader with a simple Lambert shader. Not only was the reflection sampling saved, but this also avoided the extra passes required for dynamic lights marked as ‘Important’ when using the Forward Rendering System used by Windows Mixed Reality titles.
Looking at the performance comparison above, we can see that the original cost of rendering the floor was around $\sim1.5$ ms per frame, and the cost with the replacement shader was only $\sim0.3$ ms per frame. This is a 5x performance improvement.
Figure 8: The assembly for our shader was reduced from 267 instructions (left) down to only 47 (right) and had significantly less pixel shader invocations per sample.

As shown in the figure above, the assembly for our shader was reduced from 267 instructions down to only 47 and had significantly less pixel shader invocations per sample.
Figure 9: Side-by-side comparison of the same scene, with a standard shader on the left and the optimized Lambert shader on the right.

The above image shows the high-end build with no changes except for the replacement of the standard shader with the Lambert shader. Notice that after all these reductions and cuts, we’re still left with a good-looking, cohesive floor. Microsoft has also created optimized versions of many of the Unity built-in shaders and added them as part of their Mixed Reality Toolkit. Experiment with the materials in the toolkit and see how they affect the look and performance of your game.

**Shaders—Material Batching with Unlit Shaders**

Draw call batching is the practice of bundling up separate draw calls that share common state properties into batches. The render thread is often a point of bottleneck contention, especially on mobile and VR, and batching is only one of the main tools in your utility belt to alleviate driver bottlenecks. The common properties required to batch draw calls, as far as the Unity engine is concerned, are materials and the textures used by those materials. There are two kinds of batching that the Unity engine is capable of: static batching and dynamic batching.

**Static batching** is very straightforward to achieve, and typically makes sense to use. As long as all static objects in the scene are marked as static in the inspector, all draw calls associated with the mesh renderer components of those objects will be batched (assuming they share the same materials and textures). It’s always best practice to mark all objects that will remain static as such in the inspector for the engine to smartly optimize unnecessary work and remove them from consideration for the various internal systems within the Unity engine, and this is especially true for batching. Keep in mind that for Windows Mixed Reality mainstream, instanced stereoscopic rendering is not implemented yet, so any saved draw calls will count two-fold.

**Dynamic batching** has a little bit more nuance. The only difference in requirements between static and dynamic batching is that the vertex attribute count of dynamic objects must be considered and stay below a certain threshold. Be sure to check the Unity documentation for what that threshold is for your version of Unity. Verify what is actually happening behind the scenes by taking a frame capture in the Intel GPA Graphics Frame Analyzer. See Figures 10 and 11 below for the differences in the frame visualization between a frame of *Space Pirate Trainer* with batching disabled and enabled.
Figure 10: Batching and instancing disabled; 1,300 draw calls; 1.5 million vertices total; GPU duration 3.5 ms/frame.

Figure 11: Batching and instancing enabled; 8 draw calls; 2 million vertices total; GPU duration 1.7 ms/frame (2x performance improvement).

As shown above, the amount of draw calls required to render 1,300 ships (1.5 million verts total) went from 1,300 all the way down to 8. In the batched example, we actually ended up rendering more ships (2 million vertices total) to drive the point home. Not only does this save a huge amount of time on the render thread, but it also saves quite a bit of time on the GPU by running through the graphics pipeline...
more efficiently. We actually get a 2x performance improvement by doing so. To maximize the total amount of calls batched, we can also leverage a technique called Texture Atlasing.

A Texture Atlas is essentially a collection of textures and sprites used by different objects packed into a single big texture. To utilize the technique, texture coordinates need to be updated to conform to the change. It may sound complicated, but the Unity engine has utilities to make it easy and automated. Artists can also use their modeling tool of choice to build out atlases in a way they’re familiar with. Recalling the batching requirement of shared textures between models, Texture Atlases can be a powerful tool to save you from unnecessary work at runtime, helping to get you rendering at less than 16.6 ms/frame.

Key takeaways:

- Make sure all objects that will never move over their lifetime are marked static.
- Make sure dynamic objects you want to batch have fewer vertex attributes than the threshold specified in the Unity docs.
- Make sure to create texture atlases to include as many batchable objects as possible.
- Verify actual behavior with Intel GPA Graphics Frame Analyzer.

**Shaders—LOD System for Droid Lasers**

For those unfamiliar with the term, LOD (Level of Detail) systems refer to the idea of swapping various asset types dynamically, based on certain parameters. In this section, we will cover the process of swapping out various materials depending on distance from the camera. The idea being, the further away something is, the fewer resources you should need to achieve optimal aesthetics for lower pixel coverage. Swapping assets in and out shouldn’t be apparent to the player. For *Space Pirate Trainer*, Van Welden created a system to swap out the **Unity standard shader** used for the droid lasers for a simpler shader that approximates the desired look when the laser is a certain distance from the camera. See the sample code below:
This is a very simple update loop that will check the distance of the object being considered for material swapping every 100 ms, and switch out the material if it’s over 30 units away. Keep in mind that swapping materials could potentially break batching, so it’s always worth experimenting to see how optimizations affect your frame times on various hardware levels.

On top of this manual material LOD system, the Unity engine also has a model LOD system built into the editor (access the documentation here). We always recommend forcing the lowest LOD for as many objects as possible on lower-watt parts. For some key pieces of the scene where high fidelity can make all the difference, it’s ok to compromise for more computationally expensive materials and geometry. For instance, in Space Pirate Trainer, Van Welden decided to spare no expense to render the blasters, as they are always a focal point in the scene. These trade-offs are what help the game maintain the look needed, while still maximizing target hardware—and enticing potential VR players.

**Lighting and Post Effects—Remove Dynamic Lights**

As previously mentioned, real-time lights can heavily affect performance on the GPU while the engine utilizes the forward rendering path. The way this performance impact manifests is through additional passes for models affected by the primary directional light as well as all lights marked as important in the inspector (up to the Pixel Light Count setting in Quality Setting). If you have a model that’s standing in the middle of two important dynamic point lights and the primary directional light, you’re looking at least three passes for that object.
Figure 13: Double the amount of dynamic lights at the base of weapons lit on each when they fire in *Space Pirate Trainer*, contributing 5 ms of frame time for the floor (highlighted).

In *Space Pirate Trainer*, the point-lights parented to the barrel of the gun were disabled in low settings to avoid these extra passes, saving quite a bit of frame time. Recalling the section about floor rendering, imagine that the whole floor was sent through for rendering three times. Now consider having to do that for each eye; you’d get six total draws of geometry that cover about 60 percent of the pixels on the screen.

Key takeaways:

- Make sure that all dynamic lights are removed/marked unimportant.
- Bake as much light as possible.
- Use light probes for dynamic lighting.
Post-Processing Effects

Post-processing effects can take a huge cut of your frame budget if care isn’t taken. The optimized “High” settings for Space Pirate Trainer utilize Unity’s post-processing stack, but still only take around 2.6 ms/frame on our surface target. See the image below:

![Figure 14: “High” settings showing 14 passes (reduced from much more); GPU duration of 2.6 ms/frame.](image)

The highlighted section above shows all of the draw calls involved in post-processing effects for Space Pirate Trainer, and the pop-out window shown is the total GPU duration of those selected calls—around 2.6 ms. Initially, Van Welden and the team tested the mobile bloom to replace the typically used effect, but found that it caused distracting flickering. Ultimately, it was decided that bloom should be scrapped and the remaining stylizing effects could be merged into a single custom pass using color lookup tables to approximate the look of the high-end version.

Merging the passes brought the frame time down from the previously noted 2.6 ms/frame to 0.6 ms/frame (4x performance improvement). This optimization is a bit more involved and may require the expertise of a good technical artist for more stylized games, but it’s a great trick to keep in your back pocket. Also, even though the mobile version of Bloom didn’t work for Space Pirate Trainer, testing mobile VFX solutions is a great, quick-and-easy experiment to test first. For certain scene setups, they may just work and are much more performant. Check out the frame capture representing the scene on “low” settings with the new post-processing effect pass implemented:
Figure 15: “Low” settings, consolidating all post-processing effects into one pass; GPU duration of 0.6 ms/frame.

**HDR Usage and Vertical Flip**

Avoiding the use of high-dynamic range (HDR) textures on your “low” tier can benefit performance in numerous ways—the main one being that HDR textures and the techniques that require them (such as tone mapping and bloom) are relatively expensive. There is additional calculation for color finalization and more memory required per-pixel to represent the full color range. On top of this, the use of HDR textures in Unity has the scene rendered upside down. Typically, this isn’t an issue as the final render-target flip only takes around 0.3 ms/frame, but when your budget is down to 16.6 ms/frame to render at 60 fps, and you need to do the flip once for each eye (~0.6 ms/frame total), this accounts for quite a significant chunk of your frame.
Figure 16: Single-eye vertical flip, 291 ms.

Key takeaways:

- Uncheck HDR boxes on scene camera.
- If post-production effects are a must, use the Unity engine’s Post-Processing Stack and not disparate Image Effects that may do redundant work.
- Remove any effect requiring a depth pass (fog, etc.).

Post Processing—Anti-Aliasing

Multisample Anti-Aliasing (MSAA) is expensive. For low settings, it’s wise to switch to a temporally stable post-production effect anti-aliasing solution. To get a feel for how expensive MSAA can be on our low-end target, let’s look at a capture of Space Pirate Trainer on high settings:

Figure 17: ResolveSubresource cost while using Multisample Anti-Aliasing (MSAA).

The ResolveSubresource API call is the fixed-function aspect of MSAA that determines the final pixels for render targets with MSAA enabled. We can see above that this step alone requires about 1 ms/frame. This is added to the additional work required per-draw that’s hard to quantify.

Alternatively, there are several cheaper post-production effect anti-aliasing solutions available, including one Intel has developed called Temporally Stable Conservative Morphological Anti-Aliasing (TSCMAA). TSCMAA is one of the fastest anti-aliasing solutions available to run on Intel® integrated graphics. If rendering at a resolution less than 1280x1280 before upscaling to native head mounted display (HMD) resolution, post-production effect anti-aliasing solutions become increasingly important to avoid jaggies and maintain a good experience.
Figure 18: Temporally Stable Conservative Morphological Anti-Aliasing (TSCMAA) provides an up to 1.5x performance improvement over 4x Multisample Anti-Aliasing (MSAA) with a higher quality output. Notice the aliasing (stair stepping) differences in the edges of the model.

Raycasting CPU-Side Improvements for Lasers
Raycasting operations in general are not super expensive, but when you’ve got as much action as there is in Space Pirate Trainer, they can quickly become a resource hog. If you’re wondering why we were worried about CPU performance when most VR games are GPU-bottlenecked, it’s because of thermal throttling. What this means is that any work across a System on Chip (SoC) generates heat across the entire system package. So even though the CPU is not technically the bottleneck, the heat generated by CPU work can contribute enough heat to the package that the GPU frequency, or even its own CPU frequency, can be throttled and cause the bottleneck to shift depending on what’s throttled and when.

Heat generation adds a layer of complexity to the optimization process; mobile developers are quite familiar with this concept. Going down the rabbit hole of finding the perfect standardized optimization method for CPUs with integrated GPUs has become a distraction, but it doesn’t have to. Just think about
holistic optimization as the main goal. Using general good practices on both the CPU and GPU will go a long way in this endeavor. Now that my slight tangent is over, let’s get back to the raycasting optimization itself.

The idea of this optimization is that raycast checking frequency can fluctuate based on distance. The farther away the raycast, the more frames you can skip between checks. In his testing, Van Welden found that in the worst case, the actual raycast check and response of far-away objects only varied by a few frames, which is almost undetectable at the frame rate required for VR rendering.

```csharp
private int raycastSkipCounter = 0;
private int raycastDynamicSkipAmount = 0;
private int distanceSkipUnit = 5;

public bool CheckRaycast()
{
    checkRaycastHit = false;
    raycastSkipCounter++;
    raycastDynamicSkipAmount = (int)(Vector3.Distance(playerOriginPos, transform.position) / distanceSkipUnit);
    if (raycastSkipCounter > raycastDynamicSkipAmount)
    {
        if (Physics.Raycast(transform.position, moveVector.normalized, out rh, transform.localScale.y * moveVector.magnitude * bulletspeed * Time.deltaTime * Mathf.Clamp(raycastDynamicSkipAmount, 1, 10), laserBulletLayerMask)) // never skip more than 16 frames
        {
            checkRaycastHit = true;
            Collision(rh.collider, rh.point, rh.normal, true);
        }
        raycastSkipCounter = 0;
    }
    return checkRaycastHit;
}
```

**Figure 19:** Sample code showing how to do your own raycasting optimization.

**Render at Lower Resolution and Upscale**

Most Windows Mixed Reality headsets have a native resolution of 1.4k, or greater, per eye. Rendering to a target at this resolution can be very expensive, depending on many factors. To target lower-watt integrated graphics components, it’s very beneficial to set your render target to a reasonably lower resolution, and then have the holographic API automatically scale it up to fit the native resolution at the end. This dramatically reduces your frame time, while still looking good. For instance, *Space Pirate Trainer* renders to a target with 1024x1024 resolution for each eye, and then upscales.
Figure 20: The render target resolution is specified at 1024x1024, while the upscale is to 1280x1280.

There are a few factors to consider when lowering your resolution. Obviously, all games are different and lowering resolution significantly can affect different scenes in different ways. For instance, games with a lot of fine text might not be able to go to such a low resolution, or a different trick must be used to maintain text fidelity. This is sometimes achieved by rendering UI text to a full-size render target and then blitting it on top of the lower resolution render target. This technique will save much compute time when rendering scene geometry, but not let overall experience quality suffer.

Another factor to consider is aliasing. The lower the resolution of the render-target you render to, the more potential for aliasing you have. As mentioned before, some quality loss can be recouped using a post-production effect anti-aliasing technique. The pixel invocation savings from rendering your scene at a lower resolution usually come in net positive, after the cost of anti-aliasing is considered.

Figure 21: Sample code for adjusting render-target scaling.
Render VR Hands First and Other Sorting Considerations
In most VR experiences, some form of hand replacement is rendered to represent the position of the players’ actual hand. In the case of *Space Pirate Trainer*, not only were the hand replacements rendered, but also the players’ blasters. It’s not hard to imagine these things covering a large amount of pixels across both eye-render targets. Graphics hardware has an optimization called early-z rejection, which allows hardware to compare the depth of a pixel being rendered to the existing depth value from the last rendered pixel. If the current pixel is farther back than the last pixel, the pixel doesn’t need to be written and the invocation cost of that pixel shader and all subsequent stages of the graphics pipeline are saved. Graphics rendering works like the reverse painters’ algorithm. Painters typically paint from back to front, while you can get tremendous performance benefits rendering your scene in a game from front to back because of this optimization.

![Figure 22](image)

*Figure 22:* Drawing the blasters in *Space Pirate Trainer* at the beginning of the frame saves pixel invocations for all pixels covered by them.

It’s hard to imagine a scenario where the VR hands, and the props those hands hold, will not be the closest mesh to the camera. Because of this, we can make an informed decision to force the hands to draw first. This is very easy to do in Unity; all you need to do is find the materials associated with the hand meshes, along with the props that can be picked up, and override their RenderQueue property. We can guarantee that they will be rendered before all opaque objects by using the RenderQueue enum available in the UnityEngine.Rendering namespace. See the figures below for an example.
Figure 23: RenderQueue enumeration in the UnityEngine.Rendering namespace.

```csharp
namespace UnityEngine.Rendering
{
    public enum RenderQueue
    {
        Background = 1000,
        Geometry = 2000,
        AlphaTest = 2450,
        GeometryLast = 2500,
        Transparent = 3000,
        Overlay = 4000
    }
}
```

Figure 24: Sample code for overriding a material’s RenderQueue parameter.

```csharp
using UnityEngine;
using UnityEngine.Rendering;

public class RenderQueueUpdate : MonoBehaviour
{
    public Material myVRHandsMaterial;

    // Use this for initialization
    void Start()
    {
        // Guarantee that VR hands using this material will be rendered before all other opaque geometry.
        myVRHandsMaterial.renderQueue = (int)RenderQueue.Geometry - 1;
    }
}
```

Overriding the RenderQueue order of the material can be taken further, if necessary, as there is a logical grouping of items dressing a scene at any given moment. The scene can be categorized (see figure below) and ordered as such:

1. Draw VR hands and any interactables (weapons, etc.).
2. Draw scene dressings.
3. Draw large set pieces (buildings, etc.).
4. Draw the floor.
5. Draw the skybox (usually already done last if using built-in Unity skybox).
The Unity engine’s sorting system usually takes care of this very well, but you sometimes find objects that don’t follow the rules. As always, check your scene’s frame in GPA first to make sure everything is being ordered properly before applying these methods.

**Skybox Compression**

This last one is an easy fix with some potentially advantages. If the skybox textures used in your scene aren’t already compressed, a huge gain can be found. Depending on the type of game, the sky can cover a large amount of pixels in every frame; making the sampling be as light as possible for that pixel shader can have a good impact on your frame rate. Additionally, it may also help to lower the resolution on skybox textures when your game detects it’s running on a mainstream system. See the performance comparison shown below in *Space Pirate Trainer*:
Figure 26: A 5x gain in performance can be achieved from simply lowering skybox resolution from 4k to 2k. Additional improvements can be made by compressing the textures.

Conclusion
By the end, we had *Space Pirate Trainer* running at 60 fps on the “low” setting on a 13-watt integrated graphics part. Van Welden subsequently fed many of the optimizations back into the original build for higher-end platforms so that everybody could benefit, even on the high end.

Figure 27: Final results: From 12 fps all the way up to 60 fps.

The “high” setting, which previously ran at 12 fps, now runs at 35 fps on the integrated graphics system. Lowering the barrier to VR entry to a 13-watt laptop can put your game in the hands of many more
players, and help you get more sales as a result. Download Intel GPA today and start applying these optimizations to your game.

**Resources**

Space Pirate Trainer

Intel Graphics Performance Analyzer Toolkit

TSCMAA

Unity Optimization Article

**Notices**

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products and services described may contain defects or errors known as errata which may cause deviations from published specifications. Current characterized errata are available on request.

Copies of documents which have an order number and are referenced in this document may be obtained by calling 1-800-548-4725 or by visiting www.intel.com/design/literature.htm.

This sample source code is released under the Intel Sample Source Code License Agreement.

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

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

Microsoft, Surface, and Windows are trademarks, or registered trademarks of Microsoft Corporation in the United States and/or other countries.

© 2018 Intel Corporation