Accelerate Performance Using OpenCL with Intel® Graphics Processors

Abstract
Recently, Intel announced the release of the Intel® SDK for OpenCL Applications 2013 with certified OpenCL® 1.2 support on 3rd and 4th generation Intel® Core™ processors with Intel® Iris Graphics and Intel® HD Graphics family. Visual computing applications like content creation, home movies, music, and personal images will benefit from the value of OpenCL general purpose programmability combined with access to the combined power of the CPU and the hardware acceleration capabilities of Intel® Iris Graphics and Intel® HD Graphics to increase performance and improve battery life.

In fact, with the launch of the 3rd generation Intel® Core™ processors, Intel has made it possible for software developers to heterogeneously program both CPU and Intel Graphics through the OpenCL® framework.

This article documents lessons learned while accelerating video processing with OpenCL® on the 3rd generation Intel® Core™ processors. Common video effects in a commercial application were optimized with OpenCL, and then evaluated on Intel® HD Graphics 4000. Using OpenCL, key video effects were sped up by as much as 2.3x, and with further tuning for Intel® HD Graphics 4000, additional performance gains of up to 4.3x were achieved.

Many of the lessons learned and documented in this article are also applicable to the newer versions of Intel® Graphics Processors.
Introduction

A popular software title for professional video editing was updated in 2011 to accelerate video processing effects with OpenCL. Intel used this as an opportunity to test compatibility and runtime performance of OpenCL on the Intel® 3rd Generation Core Processor with Intel HD Graphics 4000, and to determine if there were opportunities to enhance performance further. During development of the initial application release, over 60 video effects were accelerated with OpenCL, for which over 120 OpenCL kernels were implemented. With so many effects accelerated with OpenCL, it was essential to functional test and to assess performance of every OpenCL kernel.

This article outlines lessons learned and some optimization techniques used while testing and assessing the performance of the OpenCL kernels. In addition, performance and/or bottleneck issues found with some kernels and their solutions are outlined.

This article assumes the reader is familiar with the OpenCL programming model [1]. This article refers to the Intel® HD Graphics 4000 OpenCL as HDG or HDG OpenCL throughout. The HDG OpenCL capability was first introduced with the Intel 3rd Generation Core Family of Processors codenamed Ivy Bridge.
**Analysis of the Title**

A workload was released as part of the app’s release kit to highlight video processing acceleration with OpenCL. The release kit consisted of seven workloads designed to test different video effects which are accelerated with OpenCL. These seven workloads of the release kit were used in performance and functional analysis throughout the testing of the HDG implementation of OpenCL. A number of issues were encountered on the OpenCL compiler and runtime which were resolved. HDG runtime challenges were observed and it took time to understand before steps could be taken to address and optimize the performance bottlenecks observed. Figure 1 compares the initial performance to the optimized performance observed with HDG today.

The application release kit workloads were developed to determine increase in playback performance and decrease in render time. The kit is divided into 7 workloads, each showcasing different video effects. These included the following video effects, all implemented with OpenCL kernels.

**Workload 1**: includes a fade and a key framed Gaussian blur effects.

**Workload 2**: includes two crossfades, two simultaneous animated titles, a key framed black & white effect, and slow motion.

**Workload 3**: includes a key framed picture-in-picture sequence with clock wipe transition and an animated title. All of this is happening over a transition from the slow motion background image (using the Cross Effect transition) to the next clip.

**Workload 4**: includes an iris transition, a 2-track composite shot with 2 levels of secondary color correction and key framed cookie cutter effect, and a crossfade to a slow motion clip.

**Workload 5**: includes a PNG file which uses a Chroma key filter to composite it over a generated media lower third and a slow motion clip with a key framed Bump Map filter.

**Workload 6**: includes a two-track composite created by a Mask Generator filter on the top track’s event. It also features a key framed black & white effect, slow motion, and fades.

**Workload 7**: includes a PNG file with transparency over a clip with a key framed sepia effect, a key framed lens flare effect, and fades.

![Figure 1: Performance Improvements](image-url)
Figure 1 shows how early OpenCL performance compared to the performance observed with the majority of the OpenCL kernels optimized for HDG.

**General Optimizations**

This section outlines lesson learned and general optimizations for creating measureable performance improvements in OpenCL kernels. These optimizations were integrated into the release of workloads shown in Figure 1.

- Use 4x2 Chroma blocks when converting color format e.g., YUV to RGBA format. This yields as much as 20% better performance compared to processing 2x2 Chroma blocks.
- Use `fmin(a, b)` instead of `min(a, b)` for float data types.
- Use native built-in functions cautiously. Most native functions yield better performance but not all. For example: Use `y = y * y * y` when `x = 3.0f` instead of `pow(y, x)` or even `native_powr(y, 3.0f)`. The multiplication generated code is more than 2ms faster on HDG than either built-in function.
- Use bit-wise operations for Boolean comparisons whenever possible; e.g., use `if (y & x)` instead of `if (y & & x)`. This optimization improves performance especially if the kernel is big, usually 4K instructions or bigger.
- Use multiplication and truncate function `trunc(...)` to get the fractional value part instead of `fmod(x, y, 1)`. The built-in `fp = fmod(x, y, 1)` is slow compared to `fp = x * y; fp = fp − trunc(fp)`.
- Eliminate arithmetic operations of invariant variables in kernel code. Move computations to host code whenever possible. E.g., `y = a * b + c` if `c` is the only variant, compute `a * b` in host code and pass value of `a*b` in an argument parameter to the kernel.

For additional OpenCL optimization guidelines, refer to the Intel OpenCL Optimization guide.

**Performance Case Studies**

This section documents three case studies of OpenCL kernels optimized for HDG: 1) use of shared local memory, 2) use of defined filters in the HDG OpenCL engine, as well as 3) transfer and handling of texture data on HDG. The OpenCL kernels used in these studies came from a professional video editing application. The optimizations outlined in the article were scheduled to release in subsequent application updates.

**Performance Case Study 1 – Lookup Tables**

This case study documents why and how lookup tables (LUT) present performance bottlenecks in OpenCL kernels. The study examines the performance of the “color curves” video effect program where three lookup tables are used. Analysis of HDG OpenCL capabilities determined that HDG Shared Local Memory (SLM) would speed up kernels that use LUTs. This case study proves that use of HDG SLM can in fact speed up the performance of OpenCL kernels that use LUTs. The case study also shows how simple it is to further accelerate video processing using SLM in OpenCL kernels where appropriate.

**Lookup Tables in OpenCL Kernels**

Processing lookup table data in OpenCL kernels in most cases creates a performance bottleneck. This is due to the large number of data transfers for lookup table data that occur between main memory and the memory available for the OpenCL device. The more the kernel is “compute” bound (limited by the number of computations and not the number of pixels rendered), the more severe the performance impact is. This is especially important since kernels should always be designed to be “compute” bound to maximize performance as hardware becomes more capable. If an OpenCL kernel is not compute bound, the kernel program should be redesigned or the algorithm should not be programmed using OpenCL at all. In general, lookup tables should be avoided in OpenCL kernels if at
all possible because LUTs preclude a kernel from being compute bound. Avoiding lookup tables and whether the kernel is compute bound or not are topics for another paper. For now, we’ll examine how lookup tables can manifest as performance bottlenecks and what can be done to prevent bottlenecks.

Lookup tables will almost always create a performance bottleneck when the lookup table data is large, generally more than 256 bytes. As lookup table data is being accessed by hundreds of OpenCL HW threads, lots of data transfers occur between main system memory and HDG memory. Inherently, these data transfers between the two system memories incur latency and/or collisions. The data transfer latency and access hits slow the HDG OpenCL compute engine, which in turn prevents the kernel from running optimally.

OpenCL Code on Kernels with LUTs
Consider the kernel code from the “color curves” video effect in Figure 2. Notice that in addition to other parameters, the kernel also has three parameters (global pointers) for lookup tables; these are lutR, lutG, and lutB. The OpenCL keywords “__global” implies memory is used from the global memory pool which in the HDG architecture usually means cache memory. Data held in this memory has to transfer to the kernel along a slow path. The highlighted code shows LUT table data being used, where the code is indexing through the tables to retrieve LUT data. The indexes were computed based on the incoming image pixel data – not shown here. Notice there are six values to retrieve by each kernel thread from system memory. The data transfer latency impact is compounded as there are a large number of OpenCL hardware threads running for the kernel which are trying to hit the same memory address space.

```c
__kernel void ColorCurves(…, __global uchar* lutR, __global uchar* lutG, __global uchar* lutB)
{
    ...... code omitted
    uchar4 vlow = {lutR[blow.s0], lutG[blow.s1], lutB[blow.s2], 0};
    uchar4 vhigh = {lutR[bhigh.s0], lutG[bhigh.s1], lutB[bhigh.s2], 0};
    ...... code omitted
}
```

**Figure 2: Partial Code for Color Curves Video Effect**

So what can be done to avoid bottlenecks when using lookup tables? In most cases the answer is as easy as using the HDG local memory. Local memory is also referred to as SLM (Shared Local Memory) because variables which use the “__local” prefix are allocated in local memory and local memory is shared by all work-items in a work-group. For more details on OpenCL semantics refer to the [OpenCL Specification](https://www.khronos.org/registry/cl/specs/opencl-2.2.txt).

Performance Optimization
If at all possible, lookup table data should be copied to shared local memory. Using SLM prevents excessive shuttling of lookup table data between kernel threads thus greatly minimizing data transfer penalties. With the memory latency removed or minimized, the kernel compute throughput will no longer be bogged down and should show substantially better performance. The code in Figure 3 illustrates how to use SLM for lookup tables to prevent performance bottlenecks in this OpenCL kernel.
Figure 3: Partial Code with SLM for Color Curves Video Effect

- **get local size [0/1]** – each function returns the value of the local work size specified on the kernel execution. 
  When local work size is not specified, the OpenCL kernel engine selects most appropriate based on global worksize. These values remain constant across all kernel HW threads. In this case the local work size was not specified and HDG OpenCL engine selected [64, 8], so get_local_size(0) always returned 64 and get_local_size(1) returned 8.

- **get local id [0/1]** – each function returns a value within the range of the local work size. get_local_id(0) returns a value between 0 – 63, while get_local_id(1) returns a value between 0 - 7

The **barrier(CLK_LOCAL_MEM_FENCE)** call blocks all kernel HW threads until all threads copy their corresponding chunks of the global lookup table data to local memory. The kernel code still indexes through the lookup tables, but those tables now reside in local memory. As data resides in local memory, the data transfer latency is avoided which expedites the compute part of the kernel, and thus achieves much better performance. Removing the performance bottleneck on kernels that use lookup tables is often just as simple as using SLM.

Table 1 shows the metrics of performance measured on the color curves effect which was optimized to use SLM. A stand-alone application was written and a single image (1440x1080) was used to assess the performance of the OpenCL kernel for the color curves effect. The stand-alone application host code looped the execution of the OpenCL kernel 100 times. The kernel with no SLM took about 76.5 milliseconds to process the effect on the image. While the kernel optimized with SLM took only 18 milliseconds. The performance is **4.3x** faster with SLM when compared to the performance of the kernel not using SLM.

**Table 1: Performance of Color Curves with and without SLM**

```c
typedef struct {  // Lookup table data encapsulated into one buffer to minimize clSetarg(…) calls.
uchar lutR[256];
uchar lutG[256];
uchar lutB[256];
} KernelLutRGB;

__kernel void ColorCurves(..., __global uchar* lutRGB)  // kernels one lookup table buffer
{
    .... code omitted for simplicity
    __local uchar lutR[256];
    __local uchar lutG[256];
    __local uchar lutB[256];

    int init = get_local_id(0)+get_local_size(0)*get_local_id(1);
    int step = get_local_size(0)*get_local_size(1);

    for (int i = init; i < 256; i += step)
    {
        lutR[i] = lutRGB - lutR[i];
        lutG[i] = lutRGB - lutG[i];
        lutB[i] = lutRGB - lutB[i];
    }
    barrier(CLK_LOCAL_MEM_FENCE);

    .... code omitted
    uchar4 vlow = (lutR[blow.s0], lutG[blow.s1], lutB[blow.s2], 0);
    uchar4 vhigh = (lutR[bhigh.s0], lutG[bhigh.s1], lutB[bhigh.s2], 0);
    .... code omitted
}
```
Performance was measured on Intel® 3rd generation processor with HDG OpenCL and compared to the same processor running the kernel on four logical threads. For additional system details refer to the system Information found on Appendix A.

As observed in this study, using SLM clearly benefits the runtime performance of OpenCL kernels. However, using SLM comes with some restrictions. It is important to understand the OpenCL device hardware limitations and capabilities when programming OpenCL kernels to use SLM. For HDG OpenCL and as rule of thumb, do not use SLM if the LUT size is larger than 4K bytes.

Figure 4 is a screenshot of the color curve (infrared) effect, showing the output after the different colors have been computed.

<table>
<thead>
<tr>
<th>Color Curves run type</th>
<th>Elapsed Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDG OpenCL kernel with no SLM</td>
<td>76.5</td>
</tr>
<tr>
<td>HDG OpenCL kernel with SLM</td>
<td>18.0</td>
</tr>
</tbody>
</table>

Figure 4: Sample Video Output of the Color Curves Effect

Performance Case Study 2 – Software Bilinear Interpolation

The OpenCL 1.1 specification requirements support the bilinear interpolation (BLI) algorithm through implementation of the required CLK_FILTER_LINEAR filter. Based on the OpenCL specification, the CLK_FILTER_LINEAR filter can be used to achieve bilinear interpolation. The read image function using a sampler with the CLK_FILTER_LINEAR filter returns the exact equivalent of a bilinear interpolated pixel.

Early on in the effort to accelerate video effects with HDG OpenCL, the HDG BLI capability was found to produce incorrect results. Due to these incorrect results, the video editing app programmers ended up writing their own bilinear interpolation functionality in OpenCL kernel code. The incorrect results issue was promptly fixed in the HDG
graphics driver and the performance of both BLI implementations was compared. This case study shows that HDG hardware implementation for BLI achieves better performance than that of the functionality written with OpenCL.

**Kernel Pseudo Code for Bilinear Interpolation**

The code for the page peel video effect is shown in Figure 5 and it illustrates both the software implementation for BLI as well as the use of the CLK_FILTER_LINEAR to achieve bilinear interpolation with HDG hardware.

As the code shows, the software implementation of BLI relies on four calls the “image read” built-in function, it also uses arithmetic operations. In contrast, code in second row of Figure 5 shows the use of the CLK_FILTER_LINEAR filter in a sampler. The sampler is passed as an argument to the “read image” function; this serves as an indication for HDG to use hardware BLI. Note that theScaledSampler sampler is defined so that the CLK_FILTER_LINEAR filter is used, and note that with this new sampler only one call to “read image” is needed. Highlighted code in Figure 5 shows differences between the two implementations.

```c
__constant sampler_t UnscaledSampler = CLK_NORMALIZED_COORDS_FALSE | CLK_ADDRESS_CLAMP | CLK_FILTER_NEAREST;
__constant sampler_t ScaledSampler = CLK_NORMALIZED_COORDS_FALSE | CLK_ADDRESS_CLAMP | CLK_FILTER_LINEAR;

float4 swBilinearInterpolation( input, coords)
{
    float2 fVal = // derived from incoming coords
    int2 iCoords = // derived from incoming coords
    float y0 = 1.0f – fVal.y;
    float x0 = 1.0f – fVal.x;

    float4 pixel = y0 * x0 * read image( input, UnscaledSampler, iCoords ) +
                   fVal.y * x0 * read image( input, UnscaledSampler, iCoords + (int2)( 0, 1)) +
                   y0 * fVal.x * read image( input, UnscaledSampler, iCoords + (int2)( 1, 0)) +
                   fVal.y * fVal.x * read image( input, UnscaledSampler, iCoords + (int2)( 1, 1));

    return pixel;
}

__kernel void PagePeel( inputA, inputB, outDst, …)
{
    uint cx = get_global_id(0);
    uint cy = get_global_id(1);
    int2 icord = (int2)(cx, cy);

    float4 dst;
    float4 aPixel = read image( inputA, UnscaledSampler, icord);

    float fXSrc = … X progress of the effect;
    float fYSrc = … Y progress of the effect;
    float4 bPixel = swBilinearInterpolation( inputB, (float2)(fXSrc+0.5f, fYSrc+0.5f));

    dst = mix( aPixel, bPixel, …);
    write image ( outDst, coords, dst);
}
```
The OpenCL 1.1 specification documents the CL\_FILTER\_LINEAR filter implementation as a permutation that calculates the interpolated pixel result as follows:

\[
T = (1 - y) * (1 - x) * T_{i0j0} \\
+ y * (1 - x) * T_{i1j0} \\
+ (1 - y) * x * T_{i0j1} \\
+ y * x * T_{i1j1}
\]

Where \(x\) and \(y\) values are derived from the running thread ids which you get when calling get global id 0 and 1, respectively.

**Performance Optimization**

With HDG BLI capability fixed, performance of the software implementation for BLI was compared to the performance of HDG hardware implementation of BLI. As expected, HDG hardware BLI greatly outperformed the software version.

When optimizing an OpenCL kernel for performance on HDG, if software BLI is being used, the kernel code is likely not to perform optimally. The good news is that this performance issue can easily be avoided by simply using HDG own BLI instead of an OpenCL programmed implementation for BLI. BLI is commonly used in several video effects, some of which were used to gauge the performance deltas and output result consistency. The HDG BLI consistently outperformed the software BLI implementation. Depending on the video effect, HDG BLI performed between 1.25x and 1.3x compared to the software function.

Figure 6 is a screenshot of the page peel effect being used in a video transition scenario.
For those curious about the performance of the C/C++ and OpenCL implementations for the page peel effect, Table 2 shows the performance achieved in frames per second. The HDG OpenCL implementation showed a 2.5x better performance over the CPU C/C++ implementation, and achieved an additional 1.25x with use of HDG HW bilinear interpolation.

<table>
<thead>
<tr>
<th>Page peel run type</th>
<th>FPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native C/C++</td>
<td>9.8</td>
</tr>
<tr>
<td>HDG OpenCL (Software BLI)</td>
<td>24.6</td>
</tr>
<tr>
<td>HDG OpenCL (HW bilinear interpolation)</td>
<td>30.7</td>
</tr>
</tbody>
</table>

Table 2: Performance of page peel effect with software and hardware BLI

For additional test system details, refer to the system Information found on Appendix A.

**Performance Case Study 3 – Lens Flare**

Even as OpenCL best practices and optimization guidelines suggest to program kernels with as few instructions as possible, there are exceptions to this advice. This case study explores performance shortcomings of the lens flare video effect which required six kernels, and compares the performance against a monolithic one-kernel solution.

**Texture Traffic Overhead Using Multiple Kernels**

Multiple OpenCL kernels are usually viewed as an optimal design solution for video effects where multiple independent video elements are added to the video output. In practice, a single kernel would minimize texture traffic overhead and it might be a better solution in terms of performance. This case study highlights the lens flare video effect which uses six kernels. Each kernel was designed to draw a lens effect element: poly, ring, circle, diffused ray, thin ring, and sunburst onto the video output.
Depending on the lens flare effect setting, a particular kernel would be executed multiple times to draw multiple instances of the same element on the same video frame. This required taking multiple passes over the same image and thus creating texture overhead. The video frame being processed in multiple passes incurs data traffic overhead. The traffic overhead was determined to slow down the processing of the video effect. A one-kernel approach was proposed to eliminate texture traffic and improve performance. This case study outlines the performance results with the one-kernel solution.

Performance Optimization

To consolidate six kernels into one, the unique code was taken from each of six kernels and turned into six functions which are called from the main kernel. A specific function would be called within a loop to draw multiple elements as needed. Surprisingly, not all of the settings of the lens flare effect showed performance improvement with this approach. In fact, two of the settings showed minor performance degradation. Table 3 includes the performance metrics observed with the six kernels and the one-kernel implementations. The one-kernel solution sped up three of the five settings while decreasing performance of the other two settings.

<table>
<thead>
<tr>
<th>Lens Flare run type</th>
<th>6-kernels FPS</th>
<th>1-kernel FPS</th>
<th>Scale</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setting 1 – 8 elements</td>
<td>16.9</td>
<td>14.2</td>
<td>0.84</td>
</tr>
<tr>
<td>Setting 2 – 13 elements</td>
<td>12.6</td>
<td>15.7</td>
<td>1.25</td>
</tr>
<tr>
<td>Setting 3 – 9 elements</td>
<td>17.8</td>
<td>16.7</td>
<td>0.94</td>
</tr>
<tr>
<td>Setting 4 – 23 elements</td>
<td>8.9</td>
<td>14.4</td>
<td>1.62</td>
</tr>
<tr>
<td>Setting 5 – 16 elements</td>
<td>11.8</td>
<td>15.7</td>
<td>1.33</td>
</tr>
</tbody>
</table>

Table 3: One-Kernel vs. Multiple Kernel Performance Metrics

Table 3 shows that as the number of elements to draw increase, the one-kernel implementation achieves better performance. It also shows that if the number of elements to draw is less than 10, then the six kernel implementation yields better performance. At the time of this article’s publication, further optimization of the functions engaged for setting 1 and setting 3 were still being pursued. It is still possible that the one kernel per element solution might perform better even in lens flare effects with less than 10 elements.

Figure 7 is a screenshot of the video with setting 5 of lens flare effect.
### Pseudo Code for Lens Flare

The pseudo code below includes both host and OpenCL code for the six kernels as well as for the one-kernel implementations of the lens flare effect. Figure 8 shows the host code and Figure 9 shows the OpenCL kernel code. Some code is omitted to simplify and to help illustrate key code changes.

<table>
<thead>
<tr>
<th>Host code (Six Kernels)</th>
<th>Host code (One Kernel)</th>
</tr>
</thead>
<tbody>
<tr>
<td>hostCodeFunction(...) {</td>
<td>hostCodeFunction(...) {</td>
</tr>
<tr>
<td>flareElements = N Elements to draw</td>
<td>flareElements = N Elements to draw</td>
</tr>
<tr>
<td>size_t sizeBuf = sizeof (flarePropBuf);</td>
<td>for (long i = 0; i &lt; flareElements; i++) {</td>
</tr>
<tr>
<td>fPropBuf = clCreateBuffer(clContext,...., sizeBuf, &amp;flarePropBuf, ...);</td>
<td>ElemProps[i].eType = flareProp[i].eType;</td>
</tr>
<tr>
<td>for (long i = 0; i &lt; flareElements; i++) {</td>
<td>ElemProps[i].data1 = flareProp[i].data1;</td>
</tr>
<tr>
<td>cl_kernel kernel = kernelAry[i]; // multiple kernels</td>
<td>...</td>
</tr>
<tr>
<td>status = clSetKernelArg(kernel, 0, sizeof(cl_mem), &amp;src);</td>
<td>ElemProps[i].dataN = flareProp[i].dataN;</td>
</tr>
<tr>
<td>status = clSetKernelArg(kernel, 1, sizeof(cl_mem), &amp;dst);</td>
<td>}</td>
</tr>
<tr>
<td>status = clSetKernelArg(kernel, 4, sizeof(cl_mem), &amp;flarePropBuf);</td>
<td>size_t sizeBuf = sizeof (ElemProps);</td>
</tr>
<tr>
<td>size_t global[2] = {nWidth, nHeight};</td>
<td>flarePropBuf = clCreateBuffer(clContext, ...., nSizeBuf, &amp;ElemProps, &amp;status);</td>
</tr>
<tr>
<td>clEnqueueNDRangeKernel(cmdQueue, kernel, 2, NULL, global, NULL, 0, NULL, NULL);</td>
<td>}</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>
Figure 8: Partial Host Code for one-kernel and six-kernel Lens Flare Implementation

<table>
<thead>
<tr>
<th>Opencl Code – CL file with Six Kernels</th>
<th>Opencl Code - CL file with One Kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>_constant sampler_t iSampler = ...</td>
<td>_constant sampler_t iSampler = ...</td>
</tr>
<tr>
<td>_kernel</td>
<td>_kernel</td>
</tr>
<tr>
<td>void lensflare_e1 (input, output, ..., __constant KernelFlareProp* pFlareInfo)</td>
<td>void lens_flare_evals (input, output, ..., __constant KernelFlareProp* pFlareInfo)</td>
</tr>
<tr>
<td>{</td>
<td>{</td>
</tr>
<tr>
<td>int cx = get_global_id(0); int cy = get_global_id(1);</td>
<td>int cx = get_global_id(0); int cy = get_global_id(1);</td>
</tr>
<tr>
<td>int2 coord = (int2)(cx, cy);</td>
<td>int2 coord = (int2)(cx, cy);</td>
</tr>
<tr>
<td>float4 color = read image(input, iSampler, coord);</td>
<td>float4 color = read image(input, iSampler, coord);</td>
</tr>
<tr>
<td>if (within bounds) {</td>
<td>int nElems = (int)pFlareInfo-&gt;lensElemCount;</td>
</tr>
<tr>
<td>eval_color = e1Eval (...pFlareInfo);</td>
<td>// Get properties for each element to evaluate</td>
</tr>
<tr>
<td>color = apply_eval_color;</td>
<td>for (int i = 0; i &lt; nElems; i++) {</td>
</tr>
<tr>
<td>}</td>
<td>eType = (int)pFlareInfo-&gt;newElemProps[i].eType;</td>
</tr>
<tr>
<td>// write into output</td>
<td>data1 = pFlareInfo-&gt;newElemProps[i].data1;</td>
</tr>
<tr>
<td>write _pixel(output, coord, color);</td>
<td>dataN = pFlareInfo-&gt;newElemProps[i].dataN</td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>_kernel</td>
<td></td>
</tr>
<tr>
<td>void lensflare_e2 (input, output, ..., __constant KernelFlareProp* pFlareInfo)</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>int cx = get_global_id(0); int cy = get_global_id(1);</td>
<td></td>
</tr>
<tr>
<td>int2 coord = (int2)(cx, cy);</td>
<td></td>
</tr>
<tr>
<td>float4 color = read image(input, imageSampler, coord);</td>
<td></td>
</tr>
<tr>
<td>if (within bounds) {</td>
<td></td>
</tr>
<tr>
<td>eval_color = e2Eval (...pFlareInfo);</td>
<td></td>
</tr>
<tr>
<td>color = apply_eval_color;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>// write into output</td>
<td></td>
</tr>
<tr>
<td>write image(output, coord, color);</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>_kernel</td>
<td></td>
</tr>
<tr>
<td>lensflare_eN(....) { ... }</td>
<td></td>
</tr>
</tbody>
</table>

Figure 9: Partial OpenCL Kernel Code for one-kernel and six-kernel Lens Flare Implementation

Conclusion

In summary, video and image processing can be accelerated with OpenCL. Further optimization can be achieved on HDG with additional work. For most applications, excellent performance improvements will be observed with Intel® 3rd Generation Core Processors with HD Graphics running OpenCL, when compared to equivalent C/C++ code. Should the performance for a given kernel not improve as expected, consider the optimization techniques outlined in this paper. Refer to the Intel OpenCL optimization guide for detailed optimization techniques and methods to best performance tune OpenCL kernels for HDG.
Appendix A: System Information

Test system information:

**Hardware**

<table>
<thead>
<tr>
<th>System Item</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Core i7-3667U @ 2.0GHz</td>
</tr>
<tr>
<td>GPU</td>
<td>Intel® HD Graphics 4000 @ 350 MHz Core Clock</td>
</tr>
<tr>
<td>Memory</td>
<td>4 GB</td>
</tr>
<tr>
<td>Max DVMT</td>
<td>1696 MB</td>
</tr>
</tbody>
</table>

**Software**

<table>
<thead>
<tr>
<th>System Item</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS</td>
<td>Windows 8 Pro x64</td>
</tr>
<tr>
<td>Graphics Driver</td>
<td>9.18.10.3070</td>
</tr>
<tr>
<td>Video BIOS</td>
<td>2137.0</td>
</tr>
</tbody>
</table>

**Application Configuration**

<table>
<thead>
<tr>
<th>System Item</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graphics Settings</td>
<td>High</td>
</tr>
<tr>
<td>Resolution</td>
<td>1440x1080 UI Preview Display</td>
</tr>
</tbody>
</table>

Appendix B: Tools

Tools used for the analysis.

- Intel® OpenCL 2013 SDK Kernel Analyzer
- Intel® Graphics Performance Analyzer 4.0

Latest OpenCL SDK and tools can be found on the Intel® Media Server Studio\(^2\) or the Intel® Native Development Environment\(^2\) sites.
Appendix C: References


Acknowledgements

Thanks to Justin Landon, Bradley Werth, Jun De Vega, and Others for reviewing the article and for their valuable contributions and suggestions.
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, Core, and the Intel logo are trademarks of Intel Corporation in the US and/or other countries.

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

Copyright © 2012 Intel Corporation. All rights reserved. *Other names and brands may be claimed as the property of others.
Optimization Notice

**NOTE**: additional notices and disclaimers may be needed, depending on the content of the collateral. Generally, they can be found in the following areas, and must be appended to the Notices section of the paper.

**General Notices**: *Notices placed in all materials distributed or released by Intel*

**Benchmarking and Performance Disclaimers**: Disclaimers for Intel materials that use benchmarks or make performance claims.

**Technical Collateral Disclaimers**: Disclaimers that should be included in Intel technical materials that describe the form, fit or function of Intel products.

**Technology Notices**: Notices for Intel materials when the benefits or features of a technology or program are described. Note for technology disclaimers - if every product being discussed (e.g., ACER ULV) has the particular technology/feature, then you can remove the requirements statement in the disclaimer. If you have multiple technical disclaimers, you can consolidate the "your performance may vary" statements and only put in a single "your mileage may vary".