OpenCL* Optimizations Tutorial

Sample User's Guide

Intel® SDK for OpenCL* Applications - Samples

Document Number: 325672-003US

Contents

Legal Information

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.

A "Mission Critical Application" is any application in which failure of the Intel Product could result, directly or indirectly, in personal injury or death. SHOULD YOU PURCHASE OR USE INTEL'S PRODUCTS FOR ANY SUCH MISSION CRITICAL APPLICATION, YOU SHALL INDEMNIFY AND HOLD INTEL AND ITS SUBSIDIARIES, SUBCONTRACTORS AND AFFILIATES, AND THE DIRECTORS, OFFICERS, AND EMPLOYEES OF EACH, HARMLESS AGAINST ALL CLAIMS COSTS, DAMAGES, AND EXPENSES AND REASONABLE ATTORNEYS' FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PRODUCT LIABILITY, PERSONAL INJURY, OR DEATH ARISING IN ANY WAY OUT OF SUCH MISSION CRITICAL APPLICATION, WHETHER OR NOT INTEL OR ITS SUBCONTRACTOR WAS NEGLIGENT IN THE DESIGN, MANUFACTURE, OR WARNING OF THE INTEL PRODUCT OR ANY OF ITS PARTS.

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.

Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. Go to: http://www.intel.com/products/processor_number/.

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.

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

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

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

Microsoft product screen shot(s) reprinted with permission from Microsoft Corporation.

Copyright © 2010-2013 Intel Corporation. All rights reserved.

Optimization Notice
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

About Simple Optimizations Sample

SimpleOptimizations sample demonstrates simple ways of measuring the performance of OpenCL* kernels in an application. It discusses the basics of profiling and important caveats like having dedicated “warming” run. It also demonstrates several simple optimizations, Some of optimizations are rather CPU-specific (like mapping buffers), while others are more general (like using relaxed-math).

Introduction

Many ways to measure the performance of OpenCL kernels exist. For example there are host-side timing mechanisms like `QueryPerformanceCounter` or `rdtsc`. Still those ”wall-clock” measurements do not provide any insights into the costs breakdown, for example, whether actual kernel execution was fast but delayed by driver or run-time. For this purpose the sample shows how to employ the OpenCL profiling events. It also introduces several simple optimizations along with explanations.

Performance Debugging Intro

Most developers are probably already familiar with `QueryPerformanceCounter/ QueryPerformanceFrequency` API (otherwise please refer to Measure Code Sections Using The Enhanced Timer article on ISN). You also might want to use the in-line assembly combination of `RDTSC/CPUID`, or the `__rdtsc()` intrinsic, which is also available in 64-bit.

Below is an example of host-side timing routine around `clEnqueueNDRangeKernel` (error handling is omitted for simplicity):

```
LARGE_INTEGER g_PerfFrequency;
LARGE_INTEGER g_PerformanceCountNDRangeStart;
LARGE_INTEGER g_PerformanceCountNDRangeStop;
QueryPerformanceFrequency(&g_PerfFrequency);

QueryPerformanceCounter(&g_PerformanceCountNDRangeStart);
      clEnqueueNDRangeKernel(g_cmd_queue, …);
      clFinish(g_cmd_queue);// to make sure the kernel completed
QueryPerformanceCounter(&g_PerformanceCountNDRangeStop);
float seconds = (float)(g_PerformanceCountReadStop.QuadPart -
g_PerformanceCountReadStart.QuadPart)/(float)g_PerfFrequency.QuadPart);
```

Couple of things to pay attention to:

- The `clEnqueueNDRangeKernel function puts your kernel to the queue and immediately returns`
- Thus, to measure the kernel execution time, you need to explicitly sync on kernel completion via calling to the `clFinish` or `clWaitForEvents` functions.

Wrapping the Right Set of Operations

When using `QueryPerformanceCounter/QueryPerformanceFrequency` for understanding the performance of your kernel please ensure you wrapped the proper set of operations.

For example, avoid wrapping various `printf` calls, file Input/Output operations and other potentially costly and/or serializing routine.

Profiling Operations Using Profiling Events

Next piece of code measures the kernel execution via profiling events, again error handling is omitted:

```
g_cmd_queue = clCreateCommandQueue(…CL_QUEUE_PROFILING_ENABLE, NULL);
clEnqueueNDRangeKernel(g_cmd_queue,…, &perf_event);
```

```
clWaitForEvents(1, &perf_event);
cl_ulong start = 0, end = 0;

clGetEventProfilingInfo(perf_event, CL_PROFILING_COMMAND_START, sizeof(cl_ulong),
&start, NULL);
clGetEventProfilingInfo(perf_event, CL_PROFILING_COMMAND_END, sizeof(cl_ulong),
&end, NULL);

//END-START gives you hints on kind of "pure HW execution time"
//the resolution of the events is 1e-06
g_NDRangePureExecTime = (cl_double)(end - start)*(cl_double)(1e-06);
```

Important caveats:

- Set the `CL_QUEUE_PROFILING_ENABLE` property to enable the queue for profiling in the creation time
- You need to explicitly sync via `clWaitForEvents`. The reason is that device time counters (for the command being profiled) are associated with the specified event.

This way you can profile operations on both Memory Objects and Kernels. Refer to section 5.12 of the OpenCL* 1.1 standard for the detailed description of profiling events. Host-side wall-clock time with `QueryPerformanceCounter/ QueryPerformanceFrequency` API might result in longer execution times than precise measurements with profiling events. While for CPU the difference is typically negligible, for GPUs it can be substantial especially for lightweight kernels, for which various driver overheads might even dominate in execution time.

Comparing OpenCL Kernel Performance with Performance of Native Code

When comparing the OpenCL kernel performance with native code (e.g. C or SSE), make sure that you wrapped exactly the same set of operations. For example:

- Do not include program build time in the kernel execution time
- This build step can be amortized well via program pre-compilation (refer to `clCreateProgramFromBinary`)
- Track data transfers costs separately
- Also prefer data-mapping (example follows in this sample description); this is closer to the way a data is passed in native code (by pointers).

Also ensure the working set is identical for native/OpenCL code. Similarly, for correct performance comparison, access patterns should be the same (rows vs. columns, for example).

Getting Credible Performance Numbers

Run Kernel Several Times or at Least Have "Warming" Run

In the world of computing, performance conclusions are typically deduced from sufficiently large number of invocations of the same routine. Since the first iteration is almost exclusively slower than later iterations, minimum (or average, geomean, etc) value for the execution time is usually used for final projections. A simple alternative to having loop that calls your kernel zillion times is having single "warming" run as explained in this section.

"Warming" run is especially helpful for small/lightweight kernels for which one-time overheads (like some "lazy" object creations, delayed initializations and other costs potentially incurred by the OpenCL run-time) might really cost something. "Warming" run also brings data in the cache. Thus for bandwidth-limited kernels operating on the data that doesn't fit last-level cache, the "warming" run is unlikely to help.

Remember that profiling event is associated with **single** `clEnqueueNDRangeKernel` call. So if you call `clEnqueueNDRangeKernel` many times make sure to grab counters after each iteration:

```
cl_ulong start = 0, end = 0;
cl_ulong total_execution_time = 0;
for(int i=0;i<times;i++)
{
clEnqueueNDRangeKernel(g_cmd_queue,…, &perf_event);
clWaitForEvents(1, &perf_event);
clGetEventProfilingInfo(…&start);
clGetEventProfilingInfo(…&end);
total_execution_time += (end-start);
}
//averaging, etc
…
```

While with `QueryPerformanceCounter` you can measure the whole sequence at once, though on the *host*-side:

```
//first time-step
QueryPerformanceCounter(&g_PerformanceCountNDRangeStart);
for(int i=0;i<times;i++)
{
//assuming in-order queue, no need for sync
clEnqueueNDRangeKernel(g_cmd_queue…);
}
//waiting for the whole bunch of kernel calls to complete
clFinish(g_cmd_queue);
//last time-step
QueryPerformanceCounter(&g_PerformanceCountNDRangeStop);
```

Similarly you can also use `clEnqueueMarker` to measure the whole sequence on the *device*-side.

Kernels Should Run Sufficient Time

If your kernel is just a small number of instructions executed over small data set, then even infinitely-precise measurements mechanism is very unlikely to yield a reliable result. This is due to OS, cache, threading, etc influence. Having kernel run for at least 20 milliseconds is strongly recommended.

Potential Rule-of-Thumb

The bottom-line is that you need to build your performance conclusions on reproducible data. If "warming" run doesn't help and/or execution time still varies, you can try to run large number of iterations and then average the results (for time values that range too much, geomean is preferable).

Remember that kernels that are too lightweight wouldn't give you reliable data, so making them artificially heavier could give you important insights into the hotspots. Examples are adding loop *into* the kernel, or replicating its heavy pieces.

Optimization Tips

Below are simple optimizations that the sample demonstrates via respective command-line switches (described in the "Controlling the Sample" section).

Using Relaxed Math

One specific optimization is careful trading-off math precision and performance, refer to the "Trading off Accuracy and Speed of Calculations" chapter of the Performance Guide [1].

Use `native_*` and `half_*` variants for math built-ins where appropriately. Those can be really faster than higher precision variants. Similarly `-cl-fast-relaxed-math` compiler option enables faster versions for the whole file:

```
clBuildProgram(g_program, 0, NULL, "-cl-fast-relaxed-math", NULL, NULL);
```

Still to avoid any artifacts it is always recommended to test the output for any intolerable numerical inaccuracies (for example, by comparing to a "gold" reference implementation or output). Also comparing floating-point numbers should be performed with proper floating point epsilons.

Mapping Memory Objects

One specific optimization that is especially helpful for CPU OpenCL is mapping memory objects (buffers or images) instead of copying them into host memory. It is explained in section "Mapping Buffer Objects (USE_HOST_PTR)" of the product optimization guide.

To take advantage of this technique, memory object should be properly aligned. One way to achieve this is to allow run-time to do allocation via CL_MEM_ALLOC_HOST_PTR. To initialize the data with some values you need to map data first (as described below).

If your application uses a specific memory management algorithm, or if you want more control over memory allocation, the proper way would be creating memory objects with CL_MEM_USE_HOST_PTR flag that places object in the specified memory:

```
//min alignment query returns value in bits
cl_uint     min_align = 0; clGetDeviceInfo(g_dev,
CL_DEVICE_MEM_BASE_ADDR_ALIGN…,&min_align,…);
//here alignment should be in bytes
cl_float* g_pfInput = (cl_float*)_aligned_malloc(data_size, min_align/8);

const cl_mem_flags flags  = CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY;
//this code places the buffer in the mem-region pointed by g_pfInput cl_mem g_buf
= clCreateBuffer(g_context, flags, data_size, g_pfInput,…);
```

Mapping data is easy:

```
void* ptr = clEnqueueMapBuffer(g_queue, g_buf,…CL_MAP_READ,…data_size,…);
```

Make sure to unmap data to return ownership to OpenCL:

```
clEnqueueUnmapMemObject(g_queue, g_buf, ptr, 0, NULL, NULL);
```

Letting OpenCL Run-Time to Determine Optimal Size for Work-Groups

Important considerations for work-group size are explained in chapters "Work-Group Size Considerations" of the product optimization guide for both CPU and Intel® HD Graphics devices. The idea is to let the OpenCL implementation to automatically determine the optimal work-group size (sometimes referred as "local work size") for a given kernel. Simply pass NULL for a pointer to the local work size when calling clEnqueueNDRangeKernel:

```
//passing NULL for the pointer to the local work size
clEnqueueNDRangeKernel(g_queue, g_kernel, 1, NULL, globalWorkSize, NULL, 0, NULL,
&perf_event);
```

You can play with various work-group sizes or leave the decision to the run-time (refer to the **"**Controlling the Sample**"** section below).

Loading/Storing data in Greatest Chunks

As explained in chapter "«Gather4» Rule of Thumb" of the product optimization guide for Intel HD Graphics device it is particularly important to *load/store* data in the largest chunks (for example, using float4/int4). For CPU this is less important, though *processing* data by vectors is often beneficial.

In addition to the regular floating-point based version, the sample is also equipped with float4-based version, so you can compare performance of both versions on any device.

For float4-based version of kernel and CPU device, you may experiment with turning vectorizer off (uncomment vec_type_hint).

Controlling the Sample

The sample executable is a console application. It supports following optional command-line parameters and switches:

Option	Description
`-h, --help`	Show this help text and exit.
`-p, --platform number-or-string`	Selects the platform, the devices of which are used.
`-t, --type all | cpu | gpu | acc | default | <OpenCL constant for device type>`	Selects the device by type on which the OpenCL kernel is executed.
`-d, --device number-or-string`	Selects the device on which all stuff is executed.
`-s, --task-size <integer>`	Number of processed `floats`.
`-g, --work-group-size <integer>`	OpenCL work group size. Set to 0 for work group size auto selection.
`-r, --relaxed-math`	Enable `-cl-fast-relaxed-math` option for comilation.
`-u, --use-host-ptr`	Host pointers/buffer-mapping enabled.
`-f, --ocl-profiling`	Enable OpenCL event profiling.
`-w, --warming`	Additional "warming" kernel run enabled (useful for small task sizes).
`-v, --vector-kernel`	Enable "gather4" kernel version to process 4 floats by one work-item.
`-i, --internal-iterations <integer>`	Number of iterations in kernel.
`--errors <integer>`	Number of errors to print.

What's Next?

Using tools

Once you get the stable/reproducible performance numbers, the next question would be about what to optimize first.

Unless you suspect some specific parts of the kernel (for example, heavy math built-ins), we strongly recommend using VTune to determine hot-spots as described in the user's guide.

Remember that tuning the kernel itself might in turn require tweaking the run-time parameters as well, e.g. increasing size work-group once you kernel getting faster (larger work-groups would help to amortize run-time overheads). That is why the best practice is letting the run-time to decide on optimal local size as described above.

You can also check the overall CPU utilization and job distribution with Intel® Graphics Performance Analyzers.

Use Offline Compiler to inspect resulting assembly as described in the user's guide. Check whether your kernel is vectorized as you expect it to be, especially if you're trying to compare to your hand-tuned SSE.

Optimizing Kernels Pipeline

If you need to optimize kernels pipeline first measure kernels separately to find the most-time consuming one, using either `QueryPerformanceCounter` or profiling events as described in this document.

In the final pipeline version though, it is recommended to avoid calling `clFinish` or `clWaitForEvents` frequently (e.g. after each kernel invocation). Rather prefer to submit the whole sequence (to the in-order queue) and issue `clFinish` (or wait on the event) once. This would reduce host-device round-trips.

APIs Used

This sample uses the following APIs:

- `clKreateKernel`
- `clCreateContextFromType`
- `clGetContextInfo`
- `clCreateCommandQueue`
- `clCreateProgramWithSource`
- `clBuildProgram`
- `clCreateBuffer`
- `clSetKernelArg`
- `clEnqueueNDRangeKernel`
- `clGetEventProfilingInfo`
- `clEnqueueReadBuffer`
- `clReleaseMemObject`
- `clReleaseKernel`
- `clReleaseProgram`
- `clReleaseCommandQueue`
- `clReleaseContext`

Reference (Native) Implementation

Reference implementation is done in `ExecuteNative()` routine of `SimpleOptimizations.cpp` file. This is single-threaded code that performs exactly the same sequence as the OpenCL implementation.

References

- Intel SDK for OpenCL Applications – Optimization Guide
- OpenCL Specification Version 1.2 http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf
- Intel SDK for OpenCL Applications – User's Guide