OpenCL Optimizations Tutorial

Intel® OpenCL SDK Sample Documentation

Document Number: 325672-001US

Legal Information

Optimization Notice

Intel compilers, associated libraries and associated development tools may include or utilize options that optimize for instruction sets that are available in both Intel and non-Intel microprocessors (for example SIMD instruction sets), but do not optimize equally for non-Intel microprocessors. In addition, certain compiler options for Intel compilers, including some that are not specific to Intel micro-architecture, are reserved for Intel microprocessors. For a detailed description of Intel compiler options, including the instruction sets and specific microprocessors they implicate, please refer to the "Intel Compiler User and Reference Guides" under "Compiler Options." Many library routines that are part of Intel compiler products are more highly optimized for Intel microprocessors than for other microprocessors. While the compilers and libraries in Intel compiler products offer optimizations for both Intel and Intel-compatible microprocessors, depending on the options you select, your code and other factors, you likely will get extra performance on Intel microprocessors.

Intel compilers, associated libraries and associated development tools 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 Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (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.

While Intel believes our compilers and libraries are excellent choices to assist in obtaining the best performance on Intel and non-Intel microprocessors, Intel recommends that you evaluate other compilers and libraries to determine which best meet your requirements. We hope to win your business by striving to offer the best performance of any compiler or library; please let us know if you find we do not.

Notice revision #20110307

Contents

About Simple Optimizations

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).

Path

Location	Executable
`<INSTALL_DIR>samples\SimpleOptimizations`	`Win32\Release\SimpleOptimizations.exe` – 32-bit executable `x64\Release\SimpleOptimizations.exe` – 64-bit executable `Win32\Debug\SimpleOptimizations.exe` – 32-bit debug executable `x64\Debug\SimpleOptimizations.exe` – 64-bit debug executable

Introduction

There are many ways to measure performance of applications; in particular for OpenCL™ kernels. 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, e.g. 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

In this section we are following recipes from chapter 5 of [1].

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 inline assembly
combination of `RDTSC/CPUID`, or `__rdtsc()` intrinsic which is also available in 64-bit
unlike inline assembly.

Below is 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:

- `clEnqueueNDRangeKernel` just puts your kernel to the queue and immediately
  returns
- Thus to measure kernel execution time you need to explicitly sync on kernel
  completion via call to `clFinish` or `clWaitForEvents`.

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 i/o 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:

- The queue should be enabled for profiling (CL_QUEUE_PROFILING_ENABLE property) in 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, [2]. Notice that 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:

- Don't 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 (i.e. by pointers).

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

Finally make sure you're demanding the same accuracy. For example `rsqrt(x)` built-in is inherently of the higher accuracy than `__mm_rsqrt_ps` SSE intrinsic. There are 2 options for more fair performance in this particular case:

- Either equip `__mm_rsqrt_ps` in your native code with couple of additional Newton-Raphson iterations [3], to match the precision of OpenCL's `rsqrt`
- Alternatively you can use `native_rsqrt` in your kernel which would map exactly to `rsqrtps` instruction in the final assembly.
  - Yet another way to enable similar optimization for the *whole* program is using relaxed-math compilation flag (as described further).

Similarly to `rsqrt`, there are relaxed versions for `rcp`, `sqrt`, etc, refer to section 6.2 "Working with the `-cl-fast-relaxed-math` Flag" of [4].

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 (e.g. 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 4.2 of the Performance Guide [1].

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 chapter 2.7 of the Performance Guide [1]. 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 always query the value back from the run-time
size_t  workGroupSizeUsed;
clGetKernelWorkGroupInfo(g_kernel, g_device, CL_KERNEL_WORK_GROUP_SIZE,
sizeof(size_t), (void*)&workGroupSizeUsed, NULL);
```

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

Controlling the Sample

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

- `-h`, which displays help
- `-t <TaskSize>`, which allows you to set input (global) size to `<TaskSize>`
- `-l <GroupSize>`, which sets work group size (aka local size) to `<GroupSize>`
- `-a`, which enables auto-selecting of optimal work group size,
    - `[-l]` 'local size' parameter will be ignored if this option is specified

- `-r` , which enables relaxed math
- `-p`, which enables host pointers/buffer-mapping
- `-f`, which enables OpenCL profiling events for precise kernel execution time
- `-w`, which enables first "warming" kernel run.

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 (e.g. heavy math built-ins), we strongly recommend using VTune to determine hot-spots as described in section 7.1 "Working with the Intel® VTune™ Amplifier XE 2011" of the User Guide, [4].

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 GPA as detailed in section 7.3 "Tracing OpenCL Commands with the Intel® Graphics Performance Analyzers" of [4].

Use Offline Compiler to inspect resulting assembly as described in section 7.2 "Using the Intel® OpenCL SDK Offline Compiler" of the [4]. 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.

Project Structure

This sample project has the following structure:

- `SimpleOptimizations.cpp` - the host code, with OpenCL initialization and processing functions
- `SimpleOptimizations.cl` – source code of the OpenCL tone mapping kernel
- `SimpleOptimizations.vcproj` **–** Microsoft* Visual Studio* 2008 project file.

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

[1] Writing Optimal OpenCL* Code with the Intel® OpenCL SDK

The document is located in the Intel® OpenCL SDK installation directory: `<install-dir>\doc\`

[2] OpenCL Specification Version 1.1

http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf

[3] Increasing the Accuracy of the Results from the Reciprocal and Reciprocal Square Root Instructions using the Newton-Raphson Method. Intel application note (AP-803, Order Number: 243637-001).

[4] Intel(R) OpenCL SDK User Guide

This document is located in the Intel® OpenCL SDK installation directory: `<install-dir>\doc\`.