You can measure performance of applications in many ways; in particular for OpenCL* kernels. For example, you can perform such measurements using host-side timing mechanisms like `QueryPerformanceCounter` or `rdtsc`. Still those “wall-clock” measurements might not provide any insights into the actual cost breakdown. This article mostly duplicates the chapter “Performance Debugging” of the Intel® SDK for OpenCL* Applications - Optimization Guide. For code examples see the OpenCL* Optimizations Tutorial sample.

**Host-Side Timing**

Consider the following host-side timing routine around kernel call (error handling is omitted):

```c
float start = ...; // getting the first time-stamp
clEnqueueNDRangeKernel(g_cmd_queue, ...);
clFinish(g_cmd_queue); // to make sure the kernel completed
float end = ...; // getting the last time-stamp
float time = (end - start);
```

`clEnqueueNDRangeKernel` only puts a kernel to a queue and immediately returns. Thus, to measure kernel execution time, you need to explicitly synchronize on kernel completion by use of a call to `clFinish` or `clWaitForEvents`.

For an example of the `QueryPerformanceCounter` API or other host-side timing mechanism utilization refer to the [Intel® SDK for OpenCL* Applications - Optimization Guide](http://software.intel.com/en-us/articles/intel-sdk-for-opencl-applications-samples-getting-started/).

**Wrapping the Right Set of Operations**

When using any host-side routine for evaluating kernel performance, make sure you wrap the proper set of operations, for example, avoid:

- Including various `printf` calls
- File input and output operations
- and other potentially costly and serializing routine

Also profile kernel execution and data transfers separately by using OpenCL profiling events. Similarly, keep track of compilation and general initialization costs, such as creating buffers separately from the actual execution flow.

**Profiling Operations Using OpenCL* Profiling Events**

The next piece of code measures the kernel execution using OpenCL profiling events (error handling is omitted):

```c
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;

cGetEventProfilingInfo(perf_event, CL_PROFILING_COMMAND_START, sizeof(cl_ulong), &start, NULL);
cGetEventProfilingInfo(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-09 sec
g_NDRangePureExecTimeMs = (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.
- Explicitly synchronize using clWaitForEvents, as device time counters (for the command you profile) are associated with the specified event.

This way you can profile operations on memory objects and kernels. Refer to the OpenCL* 1.2 Specification for the detailed description of profiling events.

Host-side wall-clock time might return different results. For CPU the difference is typically negligible though. You can further break down the execution time into ”queue” and the ”driver” time using the CL_PROFILING_COMMAND_QUEUED and CL_PROFILING_COMMAND_SUBMIT.

### Comparing OpenCL Kernel Performance with Performance on Another Device or Native Code

When comparing OpenCL kernel performance on different devices or with native code (for example, C or Intel® SSE or Intel AVX), make sure that code patterns are as similar as possible:

- Wrap exactly the same set of operations.
- Do not include program build time in the kernel execution time as it can differ on different devices. You can amortize this build step using program pre-compilation with clCreateProgramFromBinary.
- Track data transfer costs separately, as such costs can differ, for example, between a discrete and an integrated GPU. Also prefer data mapping, which is closer to the way data is passed by pointers in native code. Refer to the Optimization Guide for more information.
- Ensure the working sets are identical. Similarly, for correct performance comparison, access patterns should be same (for example, rows and columns).
- Demand the same accuracy. For example, rsqrt(x) built-in inherently has higher accuracy than __mm_rsqrt_ps SSE intrinsic. To use the same accuracy in native code and OpenCL code, do one of the following:
  - Equip __mm_rsqrt_ps in your native code with a couple of additional Newton-Raphson iterations to match the precision of OpenCL rsqrt.
  - Use native_rsqrt in OpenCL kernel, which maps exactly to the rsqrtps instruction in the final assembly.
  - Use the relaxed-math compilation flag to enable similar accuracy for the whole program. Similarly to rsqrt, there are relaxed versions for rcp, sqrt, etc. For the full list refer to the Intel SDK for OpenCL Applications 2013 R2 - Optimization Guide.

### Getting Credible Performance Numbers

Performance measurements are done on a large number of invocations of the same routine. Since the first iteration is almost always significantly slower than the subsequent ones, the minimum value for
the execution time is usually used for final projections. Projections could also be made using other measures such as average or geometric mean of execution time.

An alternative to calling the kernel many times is to use a single "warm-up" run.

The warm-up run might be helpful for small or "lightweight" kernels, for example, the kernels with execution time less that 10 milliseconds. Specifically, it helps to amortize the following potential (one-time) costs:

- Bringing data to the cache
- "Lazy" object creation
- Delayed initializations
- Other costs incurred by the OpenCL* runtime

You need to build your performance conclusions on reproducible data. If the warm-up run does not help or execution time still varies, you can try running a large number of iterations and then average the results. For time values that range too much, use \textit{geomean}.

Consider the following:

- For bandwidth-limited kernels, which operate on the data that does not fit in the last-level cache, the "warm-up" run does not have as much impact on the measurement.
- For a kernel with a small number of instructions executed over a small data set, make sure there is a sufficient number of iterations, so the kernel runs for at least 20 milliseconds.

Kernels that are very lightweight do not give reliable data, so making them artificially heavier could give you important insights into the hotspots. For example, you can add loop in the kernel, or replicate its heavy pieces.

Refer to the OpenCL* Optimizations Tutorial SDK sample for code examples of performing the warm-up activities before starting performance measurement.

\section*{Using Tools}

Once you get reproducible performance numbers, you need to choose what to optimize first.

To optimize your OpenCL* kernel, you can use the following tools:

- Intel® VTune™ Amplifier XE 2013, which enables you to fine-tune your code for optimal OpenCL CPU and Intel Processor Graphics device performance, ensuring that hardware capabilities are fully utilized.
- Intel® SDK for OpenCL Applications - Kernel Builder, a tool, which offers full offline OpenCL language compilation, including an OpenCL syntax checker, cross hardware compilation support, Low Level Virtual Machine (LLVM) viewer, assembly language viewer and intermediate program binaries generator. It also enables experimenting with running kernels on a specific device without writing a host code.

Unless you suspect some specific parts of the kernel (for example, intensive use of math built-ins), consider using the Intel VTune Amplifier XE to determine hot-spots when executing on OpenCL CPU device. Refer to the Intel SDK for OpenCL Applications 2013 R2 - User’s Guide for more information.

Use The Intel SDK for OpenCL Applications - Kernel Builder to inspect resulting assembly. Check whether your kernel is vectorized (for OpenCL CPU) as you expect it to be, especially if you are trying to compare it to your hand-tuned Intel® SSE/Intel® AVX intrinsics.