

# Writing Optimal OpenCL™ Code with Intel® OpenCL SDK

## Performance Guide

---

Copyright © 2010–2011 Intel Corporation

All Rights Reserved

Document Number: 325696-001US

Revision: 1.3

World Wide Web: <http://www.intel.com>



## 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.

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>.

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/](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 by Khronos.

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

Copyright © 2010-2011 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



# Contents

---

1	About This Document .....	6
1.1	Prerequisites .....	6
1.2	Related Documents .....	6
2	Introduction .....	7
2.1	Basic OpenCL™ Concepts.....	7
2.2	Using Data Parallelism .....	8
2.3	Methods in Coding for Intel Architecture .....	10
2.4	Benefitting from Implicit Vectorization .....	10
2.5	Writing Kernels to Benefit From Implicit Vectorization.....	11
2.6	Using Vector Data Types.....	11
2.7	Writing Kernels to Directly Target the Intel Architecture Vector Units .....	13
2.8	Work-Group Size Considerations.....	14
2.9	Work-Group Level Parallelism .....	15
3	Optimizing OpenCL Usage.....	17
3.1	Using Memory Objects.....	17
3.2	Avoiding Needless Synchronization .....	17
3.3	Image Support .....	18
3.4	Re-using Compilation Results with <code>clCreateProgramWithBinary</code> .....	19
4	Performance Debugging Intro.....	20
4.1	Host-side timing .....	20
4.2	Wrapping the right set of operations .....	20
4.3	Profiling Operations using OpenCL Profiling Events .....	21
4.4	Comparing OpenCL Kernel Performance with Performance of Native Code.....	21
4.5	Getting Credible Performance Numbers .....	22
4.6	Using tools.....	23
5	Tips and Tricks for Kernel Development .....	24
5.1	Why Optimizing Kernel Code Is Important? .....	24
5.2	Avoiding Spurious Operations in Kernel Code .....	24
5.3	Performing Initialization in a Separate Task.....	25



5.4	Avoiding Handling Edge Conditions in Kernels.....	26
5.5	Using the Preprocessor for Constants .....	27
5.6	Prefer Signed Integer Data Types Over Unsigned.....	28
5.7	Row-wise data acceses are generally preferable.....	29
5.8	Trading off Accuracy and Speed of Calculations .....	30
5.9	Some tips for Auto-Vectorization Module.....	30
5.10	Local Memory Usage .....	31
5.11	Built-In Functions .....	32
5.12	Avoid Extracting Vector Components.....	32
6	Task-Parallel Programming Model Hints .....	34



# 1 *About This Document*

---

This document highlights general principles of efficient programming for modern Intel® architectures using OpenCL™. It focuses on CPU-specific optimization fundamentals without explaining higher level tool chain. For this information, please see the documents referred in the [Related Documents](#) section of this guide.

## 1.1 Prerequisites

This document assumes background knowledge of OpenCL. You should also be familiar with the basic concepts of Single Instruction Multiple Data (SIMD) vector instruction sets.

To get started, you can see the OpenCL specification [2] or the Overview presentation of OpenCL at <http://www.khronos.org>.

## 1.2 Related Documents

The following is a list of documents referenced in this Guide:

[1] Intel(R) OpenCL SDK User Guide

This 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] Real-Time Shallow Water Simulation with OpenCL for CPUs. GDC'11 Khronos OpenCL BOF

<http://software.intel.com/file/35807>



## 2 Introduction

---

Basically, you can achieve parallelism via concurrent work-items execution. This guide explains how you can benefit from the automatic vectorization using OpenCL on Intel CPUs. Another option to consider is using vector data types within a work-item. This approach is sometimes referred to as *explicit vectorization*.

OpenCL permits you to use both Data Parallel and Task Parallel Programming Models [2]. The following sections of this guide introduce a few OpenCL concepts, provide an overview of these programming models, and give optimization tips for each model.

### 2.1 Basic OpenCL™ Concepts

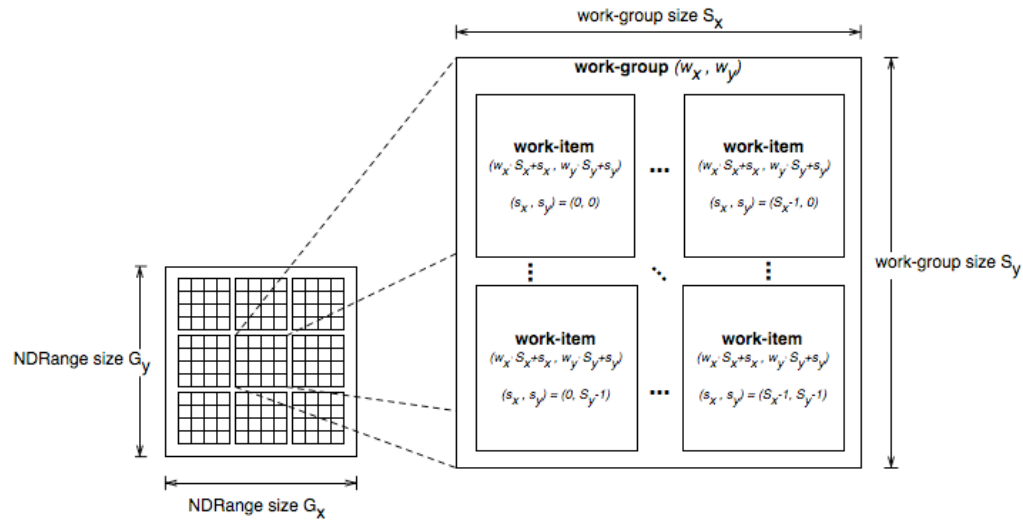
One of the main concepts in OpenCL is a *kernel* – a function declared in an OpenCL program. A running instance of a kernel code is called a *work-item*.

Each kernel is compiled for the specific *device(s)* that consist of *compute units*. Work-items are executed on the compute units by *work-groups*. Each work-group has the following properties:

- Data sharing between work-items via local memory
- Synchronization between work-items via barriers and memory fences
- Special work-group level built-in functions, such as `work_group_copy`.

A multi-core CPU constitutes a single OpenCL device. Multiple CPUs also appear a single OpenCL device. According to this logic, separate cores are compute units. The “device fission” extension (introduced with Beta release of the SDK) relaxes this model; refer to the [1] for description.

The main difference between the traditional C execution model and OpenCL execution model is that kernel instances are executed concurrently over a virtual grid defined by the *host code*. Consider the following figure [2]:



When launching the kernel for execution, the host code defines the grid dimensions, or the global work size. The host code can also define the partitioning to work-groups, or leave it to the implementation. During the execution, the implementation runs a single work-item for each point on the grid. It also groups the execution on compute units according to the work-group size.

The order of execution of work-items within a work-group, as well as the order of work-groups, is implementation-specific.

**NOTE:** Task-Parallel OpenCL programming model is also mapped to this model. The task parallel execution runs a single work-group, with a single work-item in it.

## 2.2 Using Data Parallelism

OpenCL basic data parallelism uses Single Program Multiple Data (SPMD) coding style. SPMD resembles fragment processing with pixel shaders in the context of graphics.

In this programming model, a kernel is executed concurrently on multiple elements. Each element has its own data and its own program counter. If elements are vectors of any kind (for example, 4-way pixel values for RGBA image), using vector types is a natural choice.

This section describes how to convert regular C code to an OpenCL program using a simple "hello world" style example. Consider the following C function:

```
void scalar_mul(int n, const float *a, const float *b, float *result)
{
```





```
int i;
for (i = 0; i < n; ++i)
    result[i] = a[i] * b[i];
}
```

This function performs element-wise multiplication of two arrays, `a` and `b`. Each element in `result` stores the product of the matching elements from arrays `a` and `b`.

Note the following:

- The `for` loop contains two parts: the range of operation (a single dimension containing `n` elements), and the internal parallel operation.
- The basic operation is done on scalars (floats).
- Loop iterations are independent.

In OpenCL, the same function looks as follows:

```
__kernel void scalar_mul(__global const float *a,
                        __global const float *b,
                        __global float *result)
{
    int id = get_global_id(0);
    result[id] = a[id] * b[id];
}
```

The kernel function performs the same basic element-wise multiplication of two scalars. The index is provided by using a built-in function that gives the *global ID*, a unique number for each work-item within the grid (NDRange).

Note that the code itself does not imply any parallelism. Only the combination of the code with the execution over a global grid exploits the parallelism of the device.

This parallelization method abstracts the details of the underlying hardware. You can write your code according to the native data types of the algorithm, leaving the actual mapping to specific hardware to the implementation.

Section 2.6 describes another way to write the OpenCL kernels, in which you tailor your code to the underlying hardware. This might sacrifice *performance portability* but also provides maximum performance gain on the specific platform.



## 2.3 Methods in Coding for Intel Architecture

Modern Intel processors provide acceleration using Single Instruction Multiple Data (SIMD) instruction sets that include a wide range of Intel® Streaming SIMD Extensions (Intel® SSE) and Intel® Advanced Vector Extensions (Intel® AVX) instructions.

Today CPUs support SIMD parallelism on relatively short vectors. By processing multiple data elements in a lock-step, these ISA extensions enable data parallelism usage in scientific, engineering, or graphic applications.

On Intel SIMD architecture, vector registers can store a group of data elements of the same data type, such as `float` or `char`. The number of data elements packed depends on the microarchitecture as well as on the base data type width:

- For example for the microarchitecture code-named SandyBridge the vector width is up to 256 bits. Each vector (YMM) register can store 8 floats, 8 32-bit integers, etc.

When dealing with code written in SPMD style, the OpenCL implementation can map the work-items to the hardware in one of the following ways:

- Mapping work-items to be executed via scalar code. Work-items are executed one-by-one.
- Mapping work-items to SIMD elements, as explained above. In this case, several work-items are packed together to run simultaneously.

Intel® OpenCL SDK contains an implicit vectorization module that implements the second scheme. Depending on the kernel code, this operation may have some limitations. If the vectorization module optimization is disabled, SDK uses the first scheme.

## 2.4 Benefitting from Implicit Vectorization

Intel® OpenCL SDK includes an implicit vectorization module as part of the program build process. When enabled, this module packs several work-items to be executed in a lock-step via SIMD instructions. This allows you to benefit from the vector units in Intel hardware without writing explicit vector code to specifically target them.



Vectorization module transforms scalar operations on adjacent work-items into an equivalent vector operation. When vector operations already exist in the kernel source code, they are scalarized (broken down into component operations) and re-vectorized. It provides additional performance gain by transforming the memory access pattern of the kernel into structure of arrays (SOA), which is often more cache-friendly than the array of structures (AOS).

You can find some details on the vectorization approach in [\[3\]](#).

## 2.5 Writing Kernels to Benefit From Implicit Vectorization

The implicit vectorization module works best for kernels that operate on elements of 4-byte width, such as `float` or `int`. In OpenCL, you can define the computational width of a kernel using the `vec_type_hint` attribute.

By default, kernels are always vectorized, since the default computation width is 4-byte. Specifying `__attribute__((vec_type_hint(<typen>)))` with `typen` of any vector type (e.g. `float3` or `char4`) disables the vectorization module optimization for this kernel.

A performance benefit from the vectorization module might be lower for kernels that include complex control flow which depends on the work-item ID.

To benefit from vectorization, you do not need to necessarily use `for` loops within kernels. In other words, you don't need to loop over several data elements inside a kernel. It is better to have the kernel dealing with a single data element, and let the vectorization module take care of the rest. The more straightforward your OpenCL code is, the more optimization you may get from vectorization.

Also please refer to section 5.9 for some simple tips on vectorization.

## 2.6 Using Vector Data Types

You can maximize utilization of the CPU vector units by using vector data types explicitly through the kernel code. In this style, you map vector data types directly to the hardware vector registers. Thus, the data types used should match the width of the underlying SIMD. For example, on processors based on the SandyBridge



microarchitecture you might want to use `float8`, `double4`, etc for best performance. Notice that `int8` would not give you any gains, since Intel® AVX supports full-blown vectors of width 8 for the floating-point and doubles data types only.

Notice that this way a code is tied to the specific underlying hardware (e.g. SIMD width). This might somewhat sacrifice the performance portability but gain the maximum performance on the specific platform.

You might want to use even wider data types, such as `int8` or `float16`, to transparently cover many potential SIMD widths. However, using types wider than the real underlying hardware supports is somewhat similar to loop-unrolling. This might be performance advantageous in some cases, but also increases register pressure, so some experimenting is required.

With vector data types, each work-item now processes N elements. You should reduce the size of the grid (number of work-items required to process the same dataset) accordingly by N.

In this coding style, you plan the vector-level parallelism yourself (instead of relying on the implicit vectorization module described in the following chapter). This approach can be beneficial in these scenarios:

- You are porting the code that originally used Intel® SSE/AVX instructions.
- You want to benefit from hand-tuned vectorization of your code.

The following example shows the same multiplication kernel written to target the 256-bit vector units of a processor from 2nd Generation Intel® Core™ family:

```
__kernel __attribute__((vec_type_hint(float8)))
void edp_mul(__global const float8 *a,
             __global const float8 *b,
             __global float8 *result)
{
    int id = get_global_id(0);
    result[id] = a[id] * b[id];
}
```

In this example, the data passed to the kernel represents buffers of `float8`. The calculations are performed on 8 elements together.

The attribute added before the kernel signals the compiler or implementation that this kernel is written in an optimized vectorized form, so the implicit vectorization module does not operate on it. You must use `vec_type_hint` to disable vectorization if you



kernel already processes data using mostly vector types. For more details on this attribute, see section 6.7.2 of the OpenCL Specification [\[2\]](#).

## 2.7 Writing Kernels to Directly Target the Intel Architecture Vector Units

Usage of the OpenCL vector data types permits direct utilization of the Intel vector instruction set as described in section 2.6. For instance, consider the following OpenCL snippet:

```
float4 a, b;  
float4 c = a + b;
```

After compilation, it resembles the following C snippet in intrinsics:

```
__m128 a, b;  
__m128 c = _mm_add_ps(a, b);
```

Or in assembly:

```
movaps xmm0, [a]  
addps  xmm0, [b]  
movaps [c], xmm0
```

However, in contrast to the code in intrinsics, the OpenCL kernel using `float4` would transparently benefit from Intel® AVX if compiler finds it beneficial to promote `float4` to `float8`. Auto-vectorization module is capable of packing work-items in automatic way, though it might be sometimes less efficient than manual packing.

So if the native size for your kernel requires less than 128 bits (256 bits for floating point and doubles) and you want to rely on explicit vectorization, the best solution is to manually pack work-items together.

For instance, suppose your kernel receives an  $(x, y)$  float coordinates and shifts it by  $(dx, dy)$ :

```
__kernel void shift_by(__global float2* coords, __global float2* deltas)  
{  
    int tid = get_global_id(0);  
    coords[tid] += deltas[tid];  
}
```

This kernel uses the `float2` vector type. This is less efficient compared to the version with manually packed pairs of work-items:



```
//Assuming the target is Intel® AVX enabled CPU
__kernel __attribute__((vec_type_hint(float8)))
void shift_by(__global float2* coords, __global float2* deltas)
{
    int tid = get_global_id(0);
    float8 my_coords = (float8)(coords[tid], coords[tid + 1],
                                coords[tid + 2], coords[tid + 3]);
    float8 my_deltas = (float8)(deltas[tid], deltas[tid + 1],
                                deltas[tid + 2], deltas[tid + 3]);
    my_coords += my_deltas;
    vstore8(my_coords, tid, (__global float*)coords);
}
```

Every work-item in this kernel does four times as much work as a work-item in the previous kernel. Consequently, only one fourth of invocations is required (and run-time overheads are better amortized), so you need to change the host code accordingly.

The best performance can be gained when using vectors of 32-bit data types, e.g. `int4` and `float4/float8` data types. Other types may cause a behind-the-scene upcast of the input data that has negative impact on performance.

## 2.8 Work-Group Size Considerations

We always recommend letting 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`.

If you want to experiment with workgroup sizes please take into account the following:

- To benefit from the auto-vectorization optimization (section 2.4), the work-group size must be a multiple of 8.
- The best way to fit multiple architectures is querying device for the `CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE` parameter via call to `clGetKernelWorkGroupInfo` and using the value to set work-group size accordingly.
- If the kernel code contains the barrier instruction, issuing it causes a lightweight context switch. It forces the framework to save the state of all the



work-items in the work-group prior to the barrier instruction. In this case, the issue of work-group size becomes a tradeoff - the more local and private memory each work-item in the work-group requires, the smaller the optimal work-group size might potentially be. The reason is that a barrier also issues copying for the total amount of private and local memory used by all work-items in the work-group.

- On the other side, small work-groups exhibit scheduling overhead, so the larger workgroup the better overhead is amortized. Still for the load-balancing reasons it is recommended to have sufficiently large number of work-groups (as explained in the next section). So this is a trade-off again.

Generally, the recommended work-group size for kernels is **64–128**.

## 2.9 Work-Group Level Parallelism

Consider the following loop:

```
for (int i = 0; i < 1024; ++i)
{
    data[i]++;
}
```

In OpenCL, the equivalent is to launch the following kernel as a one-dimensional **NDRange** of size 1024:

```
__kernel void inc(__global int* data)
{
    data[get_global_id(0)]++;
}
```

As the amount of work done in the loop body is small, a conventional native C compiler would unroll the loop above by a certain factor for better performance:

```
for (int i = 0; i < 256; ++i)
{
    //unrolling 4 times
    data[4 * i + 0]++;
    data[4 * i + 1]++;
    data[4 * i + 2]++;
    data[4 * i + 3]++;
}
```



This also has an OpenCL equivalent, as a kernel invocation can perform more than one work-item referred by global ID. It can iterate on several indices in the index space:

```
__kernel void inc_unrolled(__global int* data)
{
    int tid = get_global_id(0);
    data[4 * tid + 0]++;
    data[4 * tid + 1]++;
    data[4 * tid + 2]++;
    data[4 * tid + 3]++;
}
```

This method is a useful optimization for lightweight kernels. Besides, given a constant work-group size and an index space whose size cannot be controlled, it enables the host code to effectively control the number of work-groups created.

As work-groups are independent, they can be executed concurrently on different hardware threads. So, the number of work-groups should not be less than the number of logical cores (as reported by the OS). A larger number of work-groups results in more flexibility in scheduling at the cost of more task switching overhead.

To get the largest performance gain from parallelism between work-groups or different tasks, you should ensure that the execution of a work-group or a task takes around 10,000 – 100,000 instructions. A smaller value increases the proportion of switching overhead compared to actual work. However, this is a rule of thumb, so you should experiment.





## 3 *Optimizing OpenCL Usage*

---

### 3.1 Using Memory Objects

As the host code shares the physical memory with the OpenCL framework, you can either request the framework to allocate memory for the host and use that, or allocate memory yourself and share the pointer with the framework. In either case, unneeded copies are avoided.

To take advantage of this, use calls to `clEnqueueMapBuffer` and `clEnqueueUnmapBuffer` instead of calls to `clEnqueueReadBuffer` or `clEnqueueWriteBuffer`. The former are lightweight (locking and unlocking the memory objects) whereas the latter affect an actual memory copy. You can apply the same optimization for mapping/un-mapping of other memory objects, such as images.

If your application uses a specific memory management algorithm, or if you want more control over memory allocation, you can allocate a buffer and then pass the pointer at `clCreateBuffer` time with the `CL_MEM_USE_HOST_PTR` flag. However, the pointer must be aligned to a 128-byte boundary. Otherwise, the framework may perform memory copies. It is recommended to query the required memory alignment using `clGetDeviceInfo` with `CL_DEVICE_MEM_BASE_ADDR_ALIGN` token.

### 3.2 Avoiding Needless Synchronization

To achieve the optimal usage of the OpenCL framework, try to avoid explicit command synchronization primitives, such as `clEnqueueMarker/Barrier`. Explicit synchronization commands and event tracking result in cross-module round trips that decrease performance.

The less explicit synchronization commands you issue, the better CPU load and performance you can get. For example, consider a scenario when you need to wait for kernel to complete execution before reading the resulting buffer. Instead of blocking after issuing the kernel with `clFinish`, you can continue execution until you really need the first buffer with results. For this purpose, use implicit synchronization via call to blocking buffer-mapping (`clEnqueueMapBuffer` with blocking set to `CL_TRUE`).



Another way to ensure that results are ready and you can proceed is using *in-order* command queues. In this execution model, the commands in a command-queue are executed in order of submission, with each command running to completion before the next one begins. This is a typical case for straightforward processing pipeline.

You can also call `clFlush` to get things rolling, and do something useful in the host thread rather than immediately block on `clFinish` and wait for results. This approach is more effective than using `clWaitForEvents`, because `clWaitForEvents` really blocks the underlying thread, whereas `clFinish` allows the thread to participate in kernels execution. Besides, using events to control execution might be error-prone.

Finally, using blocking OpenCL API is still more effective than explicit synchronization schemes based on OS sync primitives.

Also if you are optimizing kernels pipeline first measure kernels *separately* to find the most-time consuming one. 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.

## 3.3 Image Support

Intel® OpenCL SDK supports image objects. Sampler calls are performed via software emulation. So if your legacy code uses images or depends on image-specific formats, use the fastest interpolation mode that suffices your needs. For example, many (interpolating) kernels work fine with nearest-neighbor filtering. Linear filtering causes software emulation of samplers and decreases performance.

**NOTE:** If your algorithm does not require linear data interpolation, consider using buffers instead of images.

We recommend using constant samplers, so that sampling behavior is specified in compile time and can be optimized. `CL_ADDRESS_CLAMP_NONE` is the fastest addressing mode, and prefer `CL_ADDRESS_CLAMP_TO_EDGE` over `CL_ADDRESS_CLAMP`.



## 3.4 Re-using Compilation Results with `clCreateProgramWithBinary`

To retrieve binaries generated from calls to `clCreateProgramWithSource` and `clBuildProgram`, you can call `clGetProgramInfo` with the `CL_PROGRAM_BINARIES` parameter. For performance-critical applications that typically pre-compile kernel code, you can cache the resulting binaries after the first OpenCL compilation and reuse them on subsequent executions via `clCreateProgramWithBinary`. Another way to save intermediate binaries is using the Intel® OpenCL Offline Compiler tool, as described in [\[1\]](#).



## 4 Performance Debugging Intro

---

You can measure performance of applications in many ways; in particular for OpenCL™ kernels. For example there are host-side timing mechanisms like `QueryPerformanceCounter` or `rdtsc`. Still those “wall-clock” measurements might not provide any insights into the actual cost breakdown. We start this section with discussion of OpenCL™ profiling events.

### 4.1 Host-side timing

We will not discuss `QueryPerformanceCounter` API, or other host-side timing mechanisms here. Refer to the “OpenCL Optimizations Tutorial” sample from the SDK for `QueryPerformanceFrequency` code examples. The sample is located in `<install-dir>\samples\SimpleOptimizations`.

Below is trivial host-side timing routine around kernel call (error handling is omitted for simplicity):

```
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);
```

A couple of things to pay attention to:

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

### 4.2 Wrapping the right set of operations

When using any host-side routine for evaluating the performance of your kernel, please ensure you wrapped the proper set of operations.

For example avoid including various `printf` calls, file i/o operations and other potentially costly and/or serializing routine.



## 4.3 Profiling Operations using OpenCL 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-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.
- 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 might return different results. For CPU the difference is typically negligible though.

## 4.4 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, section 3.1, this is closer to the way a data is passed in a 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, 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, refer to section 5.8.

Similarly to `rsqrt`, there are relaxed versions for `rcp`, `sqrt`, etc, refer to "Working with the `-cl-fast-relaxed-math` Flag" of [1] for the full list.

## 4.5 Getting Credible Performance Numbers

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.



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.

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, do not 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.

## 4.6 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 “Working with the Intel® VTune™ Amplifier XE 2011” of the User Guide, [\[1\]](#)

Remember that tuning the kernel itself might in turn require tweaking the run-time parameters as well, e.g. increasing work-group size 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 “Tracing OpenCL Commands with the Intel® Graphics Performance Analyzers” of [\[1\]](#).

Use Offline Compiler to inspect resulting assembly as described in section “Using the Intel® OpenCL SDK Offline Compiler” of the [\[1\]](#). Check whether your kernel is vectorized as you expect it to be, especially if you’re trying to compare to your hand-tuned SSE.

## 5 *Tips and Tricks for Kernel Development*

---

### 5.1 Why Optimizing Kernel Code Is Important?

Probably the most important optimization advice is to keep thinking about kernel code as being inside (the innermost) for-loop. Indeed, issuing kernel means it is called many times by the OpenCL run-time. Every little waste is going to be costly. If you'd move something out of the loop, move it from the kernel. Examples:

- Edge detection
- Constant branches
- Variable initialization
- Variable casts.

The following sections explain the optimizations in details.

### 5.2 Avoiding Spurious Operations in Kernel Code

As every line in the kernel code is executed many times, make sure you have no spurious instructions in the kernel code.

Spurious instructions are not always evident. Consider the following kernel:

```
__kernel void unrolled(const __global int* data, const uint dataSize)
{
    size_t tid = get_global_id(0);
    size_t gridSize = get_global_size(0);
    size_t workPerItem = dataSize / gridSize;
    size_t myStart = tid * workPerItem;
    for (size_t i = myStart; i < myStart + workPerItem; ++i)
    {
        //actual work
    }
}
```

In this kernel, the for loop is used to reduce the number of work-items and the overhead of keeping them. However, in this example every work-item recalculates the





amount of indices to iterate on, while this number is identical for all work-items. As both the size of the dataset and the `NDRange` dimensions are known before kernel launch, it is preferable to calculate the amount of work per item on the host once and then pass the result as a constant parameter.

Also using `size_t` for indices makes vectorization of indexing arithmetic less efficient. A more performance friendly way is using `int` data type, when your index fits the 32-bit integer range:

```
__kernel void unrolled(const __global int* data, const uint dataSize)
{
    int tid = get_global_id(0);
    int gridSize = get_global_size(0);
    int workPerItem = dataSize / gridSize;
    int myStart = tid * workPerItem;
    for (int i = myStart; i < myStart + workPerItem; ++i)
    ...
}
```

## 5.3 Performing Initialization in a Separate Task

You may sometimes be tempted to write the following sort of code:

```
__kernel void something(const __global int* data)
{
    int tid = get_global_id(0);
    if (0 == tid)
    {
        //Do some one-shot work, like initializing a lookup table
    }
    barrier(CLK_GLOBAL_MEM_FENCE);
    //Regular kernel code
}
```

In this example, the first branch is encountered by all the work-items although it is only relevant to one. A better solution is moving the initialization phase outside the kernel code, either to a separate kernel, or to the host code.

Also if you need to run some kernel exactly once (i.e. just for single work-item) use `clEnqueueTask` which is specially crafted for this purpose.



## 5.4 Avoiding Handling Edge Conditions in Kernels

Consider this simple smoothing filter:

```
__kernel void smooth(const __global float* input,
                    uint image_width, uint image_height,
                    __global float* output)
{
    int myX = get_global_id(0);
    int myY = get_global_id(1);
    uint neighbors = 1;
    float sum = 0.0f;
    if (myX + 2 < width)
    {
        sum += input[myY * image_width + (myX + 1)];
        ++neighbors;
    }
    if (myX > 0)
    {
        sum += input[myY * image_width + (myX - 1)];
        ++neighbors;
    }
    if (myY + 2 < image_height)
    {
        sum += input[(myY + 1) * image_width + myX];
        ++neighbors;
    }
    if (myY > 0)
    {
        sum += input[(myY - 1) * image_width + myX];
        ++neighbors;
    }
    sum += input[myY * image_width + myX];
    output[myY * image_width + myX] = sum / (float)neighbors;
}
```

Assume you have a full HD image of 1920x1080. The four edge `if` conditions are executed for every pixel, that is, roughly 2 million times. However, they are only relevant for the 6000 pixels on the image edges that make 0.2% of all the pixels. For the remaining 99.8% work-items, the edge condition check would be a waste of time. The kernel below is much more effective:

```
__kernel void smooth(const __global float* input,
                    uint image_width, uint image_height,
```



```

__global float* output)
{
    int myX = get_global_id(0);
    int myY = get_global_id(1);
    float sum = 0.0f;
    sum += input[myY * image_width + (myX + 1)];
    sum += input[myY * image_width + (myX - 1)];
    sum += input[(myY + 1) * image_width + myX];
    sum += input[(myY - 1) * image_width + myX];
    sum += input[myY * image_width + myX];
    output[myY * image_width + myX] = sum / 5.0f;
}

```

Additionally, you can either pad the image appropriately (by using larger input, for example, 1922x1082) or simply ignore the pixels on the edge (by executing the kernel on a 1918x1078 sub-region within the buffer). The current version of OpenCL also permits you to use `global_work_offset` parameter with `clEnqueueNDRangeKernel` to get a similar behavior.

**NOTE:** Using image types along with the appropriate sampler (`CL_ADDRESS_REPEAT` or `CLAMP`) also automates edge condition checks for data reads. However, due to samplers being emulated on CPU, this approach is also inefficient and results in poor performance. Refer to the [Image Support](#) section for details.

## 5.5 Using the Preprocessor for Constants

Consider the following kernel:

```

__kernel void exponentor(__global int* data, const uint exponent)
{
    int tid = get_global_id(0);
    int base = data[tid];
    for (int i = 1; i < exponent; ++i)
    {
        data[tid] *= base;
    }
}

```

The number of iterations for the inner `for` loop is determined at run time, after the kernel is issued for execution. However, you can use OpenCL dynamic compilation feature to ensure the `exponent` is known at the kernel compile time (that is still host run time). In this case, the kernel looks as follows:

```

__kernel void exponentor(__global int* data)

```



```
{
    int tid = get_global_id(0);
    int base = data[tid];
    for (int i = 1; i < EXPONENT; ++i)
    {
        data[tid] *= base;
    }
}
```

Capitalization indicates that `exponent` became a preprocessor macro.

The original version of the host code passes `exponent_val` through kernel arguments as follows:

```
clSetKernelArg(kernel, 1, exponent_val);
```

The updated version uses a compilation step:

```
sprintf(buildOptions, "-DEXPONENT=%u", exponent_val);
clBuildProgram(program, <...>, buildOptions, <...>);
```

Thus, the value of `exponent` is passed during preprocessing of the kernel code. Beside saving stack space used by the kernel, this also permits the compiler to perform optimizations, such as loop unrolling or elimination.

**NOTE:** This approach requires recompiling the program every time the value of `exponent_val` changes. If you expect to change this value often, the former approach is preferable. However, this technique is often useful for transferring parameters like image dimensions to video-processing kernels, where the value is only known at host run time, but does not change once it is defined.

## 5.6 Prefer Signed Integer Data Types Over Unsigned

Many image-processing kernels operate on `uchar` input. To avoid overflow those eight-bit input values are typically converted and processed as 16- or 32-bit integer values. In both cases it is strongly recommended to use signed data types (`shorts` and `ints`), especially in case of further conversion to floating point and back.



## 5.7 Row-Wise Data Accesses Are Generally Preferable

OpenCL enables you to submit kernels on one-, two- or three-dimensional index space. Given the choice, one-dimensional ranges are preferable, for reasons of cache locality and saving index computations.

If two- or three-dimensional range naturally fits your data dimensions, try to keep work-items scanning along rows, not columns. For example:

```
__kernel void smooth(const __global float* input,
                    uint image_width, uint image_height,
                    __global float* output)
{
    int myX = get_global_id(0);
    int myY = get_global_id(1);
    int myPixel = myY * image_width + myX;
    ...
}
```

In the example above, the first dimension is the image width and the second is the image height. The following code would be less effective:

```
__kernel void smooth(const __global float* input,
                    uint image_width, uint image_height,
                    __global float* output)
{
    int myY = get_global_id(0);
    int myX = get_global_id(1);
    int myPixel = myY * image_width + myX;
    ...
}
```

In the second code example, the image height is the first dimension and the image width is the second dimension. The inefficiency comes from the fact that the framework iterates over the first dimension of the index space before the second.

The same rule applies if each work-item calculates several elements. For best performance, make sure work-items read from consecutive memory addresses.



## 5.8 Trading off Accuracy and Speed of Calculations

OpenCL offers two basic ways to trade precision for speed:

- `native_*` and `half_*` math built-ins, that have lower precision but are faster than their un-prefixed variants
- compiler optimization options like `-cl-fast-relaxed-math` flag that allows optimizations for floating-point arithmetic for the whole OpenCL program.

For the list of other compiler options and their description please refer to [\[1\]](#). In general, while the `-cl-fast-relaxed-math` flag is a quick way to get potentially large performance gains for kernels with many math operations, it does not permit fine numeric accuracy control. That is why it is also recommended to experiment with `native_*` equivalents separately for each specific case, keeping track of the resulting accuracy.

The list of functions that have optimized versions support is provided in "Working with `cl-fast-relaxed-math` Flag" section of [\[1\]](#).

## 5.9 Some Tips for Auto-Vectorization Module

As explained in the section 2.4, the vectorization module typically performs the transformation of the kernel's memory access pattern from array of structures (AOS) to structure of arrays (SOA), which is SIMD friendly.

This transformation comes with a certain cost. Organizing the input data in SOA is important to reduce the transpose penalty.

For example the following code will suffer from transpose penalty:

```
__kernel void sum(__global float4* input, __global float* output)
{
    int tid = get_global_id(0);
    output[tid] = input[tid].x + input[tid].y + input[tid].z + input[tid].w;
}
```

While the following code will not suffer from this penalty:

```
__kernel void sum(__global float* inx, __global float* iny, __global
float* inz, __global float* inw, __global float* output)
{
```



```
int tid = get_global_id(0);
output[tid] = inx[tid] + iny[tid] + inz[tid] + inw[tid];
}
```

Also be careful when dealing with branches. Particularly, avoid data loads and stores within the statements:

```
if (...) { //condition
    x = A[i1]; // reading from A
    ... // calculations
    B[i2] = y; // storing into B
} else {
    q = A[i1]; // reading from A with same index as in first clause
    ... // different calculations
    B[i2] = w; // storing into B with same index as in first clause
}
```

The better way is below:

```
temp1 = A[i1]; //reading from A in advance
if (...) { //condition
    x = temp1;
    ... // some calculations
    temp2 = y; //storing into temporary variable
} else {
    q = temp1;
    ... //some calculations
    temp2 = w; //storing into temporary variable
}
B[i2] = temp2; //storing to B once
```

## 5.10 Local Memory Usage

One typical GPU-targeted optimization is using local memory for caching of intermediate results. For CPU all OpenCL memory objects are cached by HW, so explicit caching via local memory just introduces unnecessary (moderate) overhead.



## 5.11 Built-In Functions

OpenCL offers a library of built-in functions, including vector variants. For details, please see [\[2\]](#).

Using the built-in functions is typically more efficient than their manual implementation in OpenCL code. For example, consider this code example:

```
__kernel void Foo(const __global float* a,
                  const __global float* b,
                  __global float* c)
{
    int tid = get_global_id(0);
    c[tid] = 1/sqrt(a[tid] + b[tid]);
}
```

The following code is more effective than the previous example:

```
__kernel void Foo(const __global float* a,
                  const __global float* b,
                  __global float* c)
{
    int tid = get_global_id(0);
    c[tid] = rsqrt(a[tid] + b[tid]);
}
```

In general, math, integer, and geometric built-in functions are faster than their manually-computed counterparts. Built-in functions that do not belong to these three categories should be used carefully. For example, avoid using mul24:

```
int iSize = x*y;//not mul24(x,y);
```

## 5.12 Avoid Extracting Vector Components

Consider a following kernel:

```
__constant float4 oneVec = (float4)(1.0f, 1.0f, 1.0f, 1.0f);
__kernel __attribute__((vec_type_hint(float4)))
void inverter2(__global float4* input, __global float4* output)
{
    int tid = get_global_id(0);
    output[tid] = oneVec - input[tid];
    output[tid].w = input[tid].w;
    output[tid] = sqrt(output[tid]);
}
```





For this example of the explicitly vector code the extraction of the *w* component is very costly. The reason is that the next vector operation forces re-loading the same vector from memory. It is preferable to load a vector once and perform all changes, even to a single component, via vector operations.

In this specific case, two changes are required:

1. Modify the `oneVec` so that its *w* component is zero, causing only a sign flip in the *w* component of the input vector.
2. Use float representation to manually flip the sign bit of the *w* component back.

As a result, the kernel looks as follows:

```
__constant float4 oneVec = (float4)(1.0f, 1.0f, 1.0f, 0.0f);
__constant int4 signChanger = (int4)(0, 0, 0, 0x80000000);
__kernel __attribute__((vec_type_hint(float4)))
void inverter3(__global float4* input, __global float4* output)
{
    int tid = get_global_id(0);
    output[tid] = oneVec - input[tid];
    output[tid] = as_float4(as_int4(output[tid]) ^ signChanger);
    output[tid] = sqrt(output[tid]);
}
```

At the cost of another constant vector this implementation performs all the required operations addressing only full vectors. Notice that the same way all the computations might be performed in `float8`.



## 6 *Task-Parallel Programming Model Hints*

---

Task-parallel programming model is general-purpose. It permits you to express parallelism by enqueueing multiple tasks. You can apply this model in the following usage scenarios:

- Performing different tasks concurrently by multiple threads. If you use this scenario, choose sufficient granularity of the tasks to have good load balancing.
- Adding an extra queue (beside conventional data-parallel pipeline) for tasks that occur less frequently and in asynchronous manner, for example, some scheduled events.