OpenCL™ Developer Guide for Intel® Core™ and Intel® Xeon® Processors

Contents

About This Document

1

Apply the optimizations described in this guide using the Intel® SDK for OpenCL™ Applications, which is available with:

- Intel® Media Server Studio
- Intel® System Studio
- Intel® SDK for OpenCL™ Applications standalone version

NOTE This publication, the *OpenCL™ Developer Guide for Intel® Core™ and Intel® Xeon® Processors*, was previously known as the *OpenCL™ Optimization Guide for Intel® Xeon Phi™ Coprocessor and Intel® Xeon® Processor*.

Intel SDK for OpenCL Applications includes the *Intel® Code Builder for OpenCL™ API*. Intel Code Builder for OpenCL API is a software development tool that enables development of OpenCL applications via well-known integrated development environments, targeting the Intel® Architecture processors.

The tool supports local (host-based) and remote (target-based) development on the following platforms, IDEs, and devices:

Operating System	Host/Target	Intel® SDK for OpenCL™ Applications, standalone version	Intel® SDK for OpenCL™ Applications as part of Intel® System Studio	Intel® SDK for OpenCL™ Applications as part of Intel® Media Server Studio
Windows*	Host	Yes	Yes	-
	Target	Yes	Yes	-
Linux*	Host	Yes	Yes	Yes
	Target	Yes	Yes	Yes

To better understand which version of Intel SDK for OpenCL applications is fitting to you, check the Which Version of the Intel® SDK for OpenCL™ Applications Should I Use? page.

Legal Information

Getting Help and Support

You can get support with issues you face using the Intel® SDK for OpenCL™ Applications support forum.

For information on the product requirements, known issues, and limitations, refer to the Intel® CPU Runtime for OpenCL™ Applications - Release Notes.

Introduction

About This Document

This guide describes the optimization guidelines of OpenCL™ applications targeting the Intel® Core™ and Intel® Xeon® Processors. **In case your application targets Intel® processors with Intel® Graphics, refer to the corresponding**OpenCL™ Developer Guide for Intel® Processor Graphics.

NOTE Intel® Xeon Phi™ coprocessor based on the Intel® Many Integrated Core (Intel® MIC) Architecture is supported only on OpenCL™ Runtime version 14.2.

This guide describes three basic factors most influence performance on the multi-socket systems:

- **Threading scalability**. Multi-socket Intel Xeon systems combine many Intel® CPU cores, thus utilizing thread parallelism is critical to achieving good performance.
- **Vectorization**. Intel Xeon processors support wide vector registers and associated SIMD operations.
- **Memory bandwidth utilization**.

This guide explains, which sections of code consume most compute cycles, and provides optimization best-known methods.

For better understanding of the optimizations described in this guide, you must be familiar with the following concepts:

- The OpenCL standard
- Threading and Single Instruction Multiple Data (SIMD) vector instruction sets

See Also

OpenCL™ Standard

Basic Concepts

OpenCL™ 1.2 Specification at https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf

Overview Presentations of the OpenCL™ Standard at http://www.khronos.org/registry/OpenCL

OpenCL™ Standard

OpenCL™ (Open Computing Language) standard provides a uniformed programming environment for software developers to write portable general-purpose parallel code for high-performance computing servers, client computer systems, and other computing systems. OpenCL is developed by multiple companies through the Khronos* OpenCL committee, and Intel® is a key contributor to the OpenCL standard since its inception.

Intel® OpenCL™ implementation targets Intel® Core™ and Intel® Xeon® processors.

Currently, Intel® CPU Runtime for OpenCL™ Applications has OpenCL 1.2, 2.0, and 2.1 features enabled. For OpenCL specifications, refer to:

- For OpenCL 1.2: https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf
- For OpenCL 2.0: https://www.khronos.org/registry/OpenCL/specs/opencl-2.0.pdf
- For OpenCL 2.0 C: https://www.khronos.org/registry/OpenCL/specs/opencl-2.0-openclc.pdf
- For OpenCL 2.1: https://www.khronos.org/registry/OpenCL/specs/2.2/pdf/OpenCL_API.pdf

See Also

Basic Concepts

OpenCL™ 1.2 Specification at https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf

Overview Presentations of the OpenCL™ Standard at http://www.khronos.org/registry/OpenCL

Basic Concepts

The list below explains basic OpenCL™ concepts used in this document. The concepts are based on definitions in Khronos* OpenCL specification.

- **Intel® CPU Runtime for OpenCL™ Applications** enables OpenCL software technology support on Intel® Core™ and Intel® Xeon® Processors.

  NOTE Intel® CPU Runtime for OpenCL™ Applications was previously known as Intel® SDK for OpenCL™ - CPU only Runtime Package.

- **Intel® SDK for OpenCL™ Applications** is a software development tool that enables developing, debugging, and analyzing OpenCL applications targeting the Intel® Architecture processors with the Intel® Processor Graphics.
- **An OpenCL execution model** is a principle of a kernel execution on target OpenCL device, defined by the host.
- **An OpenCL standard** is the standard for parallel programming of modern processors.
- **A compute unit** is composed of one or more processing elements and local memory. It may also include dedicated texture filter units that can be accessed by its processing elements. An OpenCL device has one or more compute units. A work-group executes on a single compute unit.
- **A device** is a collection of compute units.
- **A command-queue** is used to queue commands to a device. Examples of commands include executing kernels, or reading and writing memory objects.
- **A kernel** is a function declared in a program and executed on an OpenCL device. A kernel is identified by the `__kernel` or `kernel` qualifier applied to any function defined in a program.
- **A work-item** is one of a collection of parallel executions of a kernel invoked on a device by a command. A work-item is executed by one or more processing elements as a part of a work-group executing on a compute unit. A work-item is distinguished from other executed work-items within the collection by its global ID and local ID.

- **A work-group** is a collection of related work-items that execute on a single compute unit. The work-items in the group execute the same kernel and share local memory and work-group barriers. Each work-group has the following properties:
    - Data sharing between work-items by use of local memory
    - Synchronization between work-items by use of barriers and memory fences
    - Special work-group level built-in functions, such as `work_group_copy`

    A multi-core CPU or multiple CPUs (in a multi-socket machine) constitute a single CPU OpenCL device. Separate cores are compute units. For information on controlling the affinity by compute units using the device fission feature, refer to the OpenCL™ Device Fission for CPU Performance article.
- **A task-parallel programming model** is the OpenCL programming model that runs a single work-group with a single work item.

See Also

Using Data Parallelism

OpenCL™ 1.2 Specification at https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf

Developer Guide for Intel® SDK for OpenCL™ Applications

Using Data Parallelism

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

According to the SPMD technique, a kernel executes concurrently on multiple elements. Each element has its own data and its own program counter. If elements are vectors of any kind (for example, four-way pixel values for an RGBA image), consider using vector types.

Consider the following code example to understand how to convert a regular C code to an OpenCL code:

```
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 statement consists of two parts:
    - Range of operation (a single dimension containing `n` elements)
    - Internal parallel operation.
- The basic operation is done on scalar variables (`float` data types).
- Loop iterations are independent.

The same function in OpenCL appears as follows:

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

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

The code itself does not imply any parallelism. Only the combination of the code with the execution over a global grid implements 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. The implementation takes care of the actual mapping to specific hardware.

You can bind your code to the underlying hardware. This method provides maximum performance on a specific platform. However, performance on other platforms might be less than optimal. See the Using Vector Data Types section for more information (for CPU device only).

Check-list for OpenCL™ Optimizations

- Use Array Notation with int32 Indices: A[i][j]
- Use Floating Point for Calculations
- Note on Local Memory Use
- Use Branching Accurately
- Map Memory Objects (USE_HOST_PTR)
- Prefer Buffers Over Images
- Use Lower Math Precision
- Use Restrict Qualifier for Kernel Arguments

See Also

OpenCL™ Device Fission for CPU Performance

Use Array Notation with int32 Indices: A[i][j]

Prefer the array notation with indices of the signed `int32` type over other memory access formats. Avoid explicit pointer arithmetic like (A+1) as it results in expensive unsigned `int64` calculations and prevents various compiler optimizations. To gain performance, access any array through its unchanged base pointer using array notation, for example, A[1].

Use Floating Point for Calculations

Intel® Xeon® processors significantly accelerate floating-point calculations on the device.

Consider the following code snippet that performs calculations in `int`:

```
__kernel void scale (__constant uchar* srcA, __constant uchar* srcB, __constant uchar
nSaturation, __global uchar* dst)
        int offset = get_global_id();
        uint tempSrcA = convert_uint(srcA[offset]);//Load one RGBA8 pixel
        uint tempSrcB = convert_uint(srcB[offset]);//Load one RGBA8 pixel
        //some processing
        uint tempDst = (tempSrcA - tempSrcB) * nSaturation;
        //store
        dst[offset] = convert_uchar(tempDst);
}
```

The following example uses the `float` equivalent:

```
__kernel void scale (__constant uchar* srcA, __constant uchar* srcB, __constant uchar
nSaturation, __global uchar* dst)
```

```
        int offset = get_global_id();
        float tempSrcA = convert_float(srcA[offset]);//Load one RGBA8 pixel
        float tempSrcB = convert_float(srcB[offset]);//Load one RGBA8 pixel
        //some processing
        float tempDst = (tempSrcA - tempSrcB) * nSaturation;
        //store
        dst[offset] = convert_uchar(tempDst);
}
```

Using built-in functions improves performance. See the Use Built-In Functions section for more information.

NOTENOTE: The compiler is capable of automatic fusion of multiplies and adds. Use the `-cl-mad-enable` compiler flag to enable this optimization when compiling. Still, using explicit "mad" built-in ensures that the built-in is mapped directly to the efficient instruction.

Note on Local Memory Use

One way to optimize OpenCL™ kernels is to use local memory for caching of intermediate results. For Intel® processors, all OpenCL memory objects are cached by hardware, so explicit caching by use of local memory only introduces unnecessary (moderate) overhead.

Use Branching Accurately

You can improve the performance of the Intel® Core™ and Intel® Xeon® processors by converting the uniform conditions that are equal across all work-items into compile time branches.

According to this approach, you have a single kernel that implements all desired behaviors, and let the host logic disable the paths that are not currently required. However, setting constants to branch on calculations wastes the device facilities, as the data is still being calculated before it discarded. Consider a preprocessor directives-based approach instead - use `#ifndef` blocks.

Consider the example where the original kernel uses constants for branching:

```

__kernel void foo(__constant int* src,
                 __global int*
dst,
                 unsigned char bFullFrame, unsigned char bAlpha)
{
        …
        if(bFullFrame)//uniform condition (equal for all work-items
        {
        …
                if(bAlpha) //uniform condition
                {
                …
                }
                else
                {
                …
                }
        else
        {
        …
        }
}
```

Now consider the same kernel, but with use of compile time branches ("specialization" technique):

```
__kernel void foo(__constant int* src,
                 __global int* dst)
{
        …
        #ifdef bFullFrame
        {
        …
                #ifdef bAlpha
                {
                …
                }
                #else
                {
                …
                }
                #endif
        #else
        {
        …
        }
        #endif
}
}
```

Also consider similar optimization for other constants.

Minimize or, in best case, avoid using branching in short computations with `min`, `max`, `clamp`, and select built-ins instead of `if` and `else` clauses.

Move memory accesses that are common to the `then` and `else` blocks outside of the conditional code.

Consider the original code with use of the `if` and `else` clauses:

```
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
}
```

Now consider the optimized code that uses temporary variables:

```
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
```

Mapping Memory Objects (USE_HOST_PTR)

As the host code shares the physical memory with CPU OpenCL™ device, you can do one of the following to avoid unnecessary copies:

- Request the framework to allocate memory on the host.
- Allocate properly aligned memory yourself and share the pointer with the framework.
- Use calls to `clEnqueueMapBuffer` and `clEnqueueUnmapBuffer` instead of calls to `clEnqueueReadBuffer` or `clEnqueueWriteBuffer`.

If your application uses a specific memory management algorithm, or if you need 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 the certain boundary. Otherwise, the framework may perform memory copies. Consider to query the required memory alignment using `clGetDeviceInfo` with `CL_DEVICE_MEM_BASE_ADDR_ALIGN` token.

You can also map image objects (for CPU OpenCL device), creating them with the `CL_MEM_USE_HOST_PTR` flag.

See Also

OpenCL™ 1.2 Specification at https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf

Overview Presentations of the OpenCL™ Standard at https://www.khronos.org/registry/OpenCL

Prefer Buffers over Images

On the Intel® Architecture processors, device buffers usually perform better than images, as buffers provide more data per read and write operations for buffers with much lower latency. The reason is that images are software-emulated on the CPU device. So, if your legacy code uses images or depends on image-specific formats, choose the fastest interpolation mode that suffices your needs, for example:

- Nearest-neighbor filtering, which works well for most interpolating kernels
- Linear filtering, which might decrease the CPU device performance

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

See Also

Use Floating Point for Calculations

Use Lower Math Precision

OpenCL™ offers two basic ways to trade precision for speed:

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

In general, while the `-cl-fast-relaxed-math` flag is a quick way to get performance gains for kernels with many math operations, it does not permit fine numeric accuracy control. Consider experimenting with the `native_*` equivalents separately for each specific case, keeping track of the resulting accuracy.

`Native_` versions of math built-ins are supported in hardware and run substantially faster, while offering lower accuracy. Use native trigonometry and transcendental functions, such as `sin`, `cos`, `exp`, and `log`, when performance is more important than precision.

For a full list of OpenCL build options and option descriptions, refer to the the OpenCL specification.

See Also

Use Restrict Qualifier for Kernel Arguments

Intel Tools for OpenCL™ Applications

OpenCL™ 1.2 Specification at https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf

Use Restrict Qualifier for Kernel Arguments

Consider using `restrict` type qualifier (defined by the C99) for kernel arguments (pointers) in the kernel signature. You can use the `restrict` qualifier only with kernel arguments. The qualifier is a hint to the compiler that helps to limit the effects of pointer aliasing, while also aiding caching optimizations. In the example below, it enables the compiler to assume that pointers `a`, `b`, and `c` point to the different locations. You must ensure that the pointers do not point to overlapping locations.

```
__kernel void foo(    const  float i,
                    __global const float* restrict a,
                     __global const float* restrict b,
                     __global float* restrict result)
{
//…
}
```

Tips and Tricks for Kernel Development

Why Optimizing Kernels Is Important?

As the OpenCL™ runtime calls an issued kernel many times, optimizing the kernel can bring performance gains. If you move a piece of code out of the innermost loop in a typical native code, move it from the kernel as well. Examples of code types that you might move out of the loop:

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

Avoid Spurious Operations in Kernels

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

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

```
__kernel void foo(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.

The size of the dataset and the NDRange dimensions is known before kernel launch. Calculate the amount of work per item on the host once and then pass the result as a constant parameter.

Using `size_t` for indices makes vectorization of indexing arithmetic less efficient. To improve performance, use the `int` data type, when your index fits the 32-bit integer range. Consider the following example:

```
__kernel void foo(const __global int* data, const uint workPerItem)
{
  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)="" …=""
```

See Also

OpenCL™ 1.2 Specification at https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf

Overview Presentations of the OpenCL™ Standard at https://www.khronos.org/registry/OpenCL

Avoid Handling Edge Conditions in Kernels

To understand how to avoid handling edge conditions in kernels, consider the following smoothing 2x2 filter:

```
__kernel void smooth(const __global float* input,
                     __global float* output)
{
  const int myX = get_global_id(0);
  const int myY = get_global_id(1);
  const int image_width = get_global_size(0);
  uint neighbors = 1;
  float sum = 0.0f;
  if ((myX + 1) < (image_width-1))
  {
    sum += input[myY * image_width + (myX + 1)];
    ++neighbors;
  }
  if (myX > 0)
  {
    sum += input[myY * image_width + (myX - 1)];
    ++neighbors;
  }
  if ((myY + 1) < (image_height-1))
  {
    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 that you have a full HD image with size of 1920x1080 pixels. The four edge `if` conditions are tested for every pixel, that is, roughly two million times.

However, they are only relevant for the 6000 pixels on the image edges, which is 0.2% of the pixels. For the remaining 99.8% work-items, the edge condition check is a waste of time. See the following optimized code:

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

This code requires padding (enlarging) the input buffer appropriately, while using the original global size. This way querying the neighbors for the border pixels does not result in buffer overrun. When you process several frames this way, do this padding exactly once (which means you initially allocate frames of the proper size), instead of copying each frame to larger buffer before processing, and then copying the result back to smaller (original size) buffer.

If padding through larger input is not possible, make sure you use the `min` or `max` built-ins, so checking a work-item does not access outside the actual image adds only four lines. Consider the following example:

```
__kernel void smooth(const __global float* input,
                     __global float* output)
{
  const int image_width = get_global_size(0);
  const int image_height = get_global_size(0);
  int myX = get_global_id(0);
  //since for myX== image_width-1 the (myX+1) is incorrect
  myX =  min(myX, image_width -2);
  //since for myX==0 the (myX-1) is incorrect
  myX =  max(myX, 1);
  int myY = get_global_id(1);
  //since for myY== image_height-1 the (myY+1) is incorrect
  myY =  min(myY, image_height -2);
  //since for myY==0 the (myY-1) is incorrect
  myY =  max(myY - 1, 0);
  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;
}
```

At a cost of duplicating calculations for border work-items, this code avoids testing for the edge conditions, which is otherwise needed to perform for all work-items.

See Also

Use the Preprocessor for Constants

Prefer Buffers over Images

Use the Preprocessor for Constants

Consider the following example of using the preprocessor for constants:

```
__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 the OpenCL™ dynamic compilation feature to ensure the exponent is known at kernel compile time, which is done during the host run time. In this case, the kernel appears 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. Besides saving stack space used by the kernel, this also enables the compiler to perform optimizations, such as loop unrolling or elimination. 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.

NOTENOTE: This approach requires recompiling the program every time the value of `exponent_val` changes. Prefer the variable-passing approach if you expect to change this value often.

See Also

Prefer (32-bit) Signed Integer Data Types

Prefer (32-bit) Signed Integer Data Types

Many image-processing kernels operate on `uchar` input. To avoid overflows, those eight-bit input values are typically converted and processed as 16- or 32-bit `integer` values. Use signed data types (`shorts` and `ints`) in both cases to enable the compiler to utilize a larger set of SIMD instructions.

Using `size_t`, which is another unsigned type for indices, makes the vectorization of indexing arithmetic less efficient. To improve performance, use the `int` data type for work-item parameters and loop counter, when your index fits the 32-bit integer range. Consider the following example:

```
__kernel void foo(__constant int* data, const uint workPerItem)
{
  int tid = get_global_id(0);
  int gridSize = get_global_size(0);
  for (int i = myStart; i <mystart += workperitem; ++i) …
```

Also when the compiler generates the scatter or gather instructions on non-consecutive memory accesses, it needs to safely cast to the `int32` since gather and scatter instructions use the `int32` indices. Explicit casting of the indices to the `int32` in a kernel simplifies the compiler job.

See Also

Prefer Row-Wise Data Accesses

Prefer Row-Wise Data Accesses

OpenCL™ enables you to submit kernels on one-, two-, or three-dimensional index space. Consider using one-dimensional ranges 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, the following code is not optimized (it might trigger gathers instructions):

```
__kernel void smooth(__constant float* input,
                     uint image_width, uint image_height,
                     __global float* output)
{
  int myX = get_global_id(1);
  int myY = get_global_id(0);
  int myPixel = myY * image_width + myX;
  float data = input[myPixel];
  …
}
```

In this code example, the image height is the first dimension and the image width is the second dimension. The resulting column-wise data access is inefficient, since Intel® OpenCL™ implementation initially iterates over the first dimension.

Below is more optimal version, because of more memory-friendly (sequential) access.

```
__kernel void smooth(__constant 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;
  float data = input[myPixel];
  …
}
```

In the example above, the first dimension is the image width and the second is the image height.

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

Finally, if you run two-dimensional NDRange, prefer the data access to be consecutive along dimension zero.

Use Built-In Functions

OpenCL™ offers a library of built-in functions, including vector variants. For details, see the OpenCL specification.

Using built-in functions is typically more efficient than using their manual implementation in OpenCL code. Consider the following 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 uses the `rsqrt` built-in to implement the same 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]);
}
```

Consider simple expressions and built-ins based equivalents below:

```
dx * fCos + dy * fSin == dot( (float2)(dx, dy),(float2)(fCos, fSin))
x * a - b  == mad(x, a, -b)
sqrt(dot(x, y)) == distance(x,y)
```

Use specialized built-in versions like `math`, `integer`, and `geometric` built-ins, where possible, as the specialized built-ins work faster than their manually-computed counterparts. For example, when the `x` value for `xy` is ≥0, use `powr` instead of `pow`.

See Also

The OpenCL™ 1.2 Specification at https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf

Avoid Extracting Vector Components

Consider the 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 explicit vector code, the extraction of the `w` component is very costly. The reason is that the next vector operation forces reloading the same vector from memory. Consider loading a vector once and performing all changes by use of vector operations even for a single component.

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 appears 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. All the computations might also be performed in `float8`.

Task-Parallel Programming Model Hints

Task-parallel programming model is the general-purpose that enables you to express parallelism by enqueuing multiple tasks. You can apply this model in the following scenarios:

- Performing different tasks concurrently by multiple threads. If you use this scenario, choose sufficient granularity of the tasks to enable optimal 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.

If your tasks are independent, consider using out-of-order queue.

Common Mistakes in OpenCL™ Applications

This topic describes several cases of OpenCL™ applications undefined behavior that you might encounter if you do not follow the OpenCL™ specification.

Infinite Execution of OpenCL™ Kernels

- **Independent forward progress between work items within a workgroup**

  Compilation of kernels assuming independent forward progress of the work-items may produce non-terminating code.

  Consider the following example:

```
__kernel void kern(__local uint *flag) {
    size_t id = get_local_id(0);
    if (id==0) {
        flag[0] = 0;
    }
    barrier( CLK_LOCAL_MEM_FENCE );
    while ( flag[0]<1 ) {
        if (id==0) {
            atomic_inc( &flag[0] );
        }
    }
}
```

  According to the OpenCL 2.0 specification, section 3.2.2:

  "In the absence of work-group functions (e.g. a barrier), work-items within a workgroup may be serialized. In the presence of work-group functions, work-items within a workgroup may be serialized before any given work-group function, between dynamically encountered pairs of work-group functions and between a work-group function and the end of the kernel."

  For example, if the order of work-items execution is 3, 2, 1, 0 and the work items are serialized, the first executed work item with id = 3 will never exit the loop.

  The section 3.2.2 also states:

  "The work-items within a single work-group execute concurrently but not necessarily in parallel (i.e. they are not guaranteed to make independent forward progress)."

  Redesign your code to comply with specification and not to rely on independent forward progress of work items within a work group.
- **barrier() divergence**

  Compilation of kernels containing divergent barriers may produce non-terminating code.

  A branch is considered divergent when some work items within a work group take it and other do not take it.

  A simplified code looks as follows:

```
__kernel void kern() {
  size_t lid = get_local_id(0);
  if (lid %2) { // any condition that would force only some work items to take the branch
    ...
    barrier();
  }
}
```

  According to the OpenCL™ 1.2 specification, section 3.4.3:

  "Note that the work-group barrier must be encountered by all workitems of a work-group executing the kernel or by none at all".

  It is harder to follow this rule if you place a barrier inside a loop. Redesign your code to avoid hitting the barrier only by a subset of work items within a work group.

See Also

OpenCL 1.2 Specification at https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf

OpenCL 2.0 Specification at https://www.khronos.org/registry/OpenCL/specs/opencl-2.0.pdf

Application-Level Optimizations

Avoid Needless Synchronization

For better results, avoid explicit command synchronization primitives, such as `clEnqueueMarker` and `Barrier`. Explicit synchronization commands and event tracking result in cross-module round trips, which decrease performance. The less you use explicit synchronization commands, the better the performance is.

Use the following techniques to reduce the explicit synchronization:

- Merge kernels whenever possible. It also improves data locality.
- If you need to wait for a kernel to complete execution before reading the resulting buffer, continue execution until you need the first buffer with results.
- If an in-order queue expresses the dependency chain correctly, use it to define a string of dependent kernels. In the in-order execution model, the commands in a command queue are executed in the order of submission, with each command running to completion before the next one begins. This is a typical case for a straightforward processing pipeline. Consider the following:
  - Using the blocking OpenCL™ API is more effective than explicit synchronization schemes based on OS synchronization primitives.
  - If you are optimizing the kernel pipeline, first measure kernels separately to find the most time-consuming one. Avoid calling `clFinish` or `clWaitForEvents` in the final pipeline version frequently after, for example, each kernel invocation. Prefer submitting the whole sequence (to the in-order queue) and issue `clFinish` once or wait on the OpenCL event object, which reduces host-device round trips.

See Also

Reuse Compilation Results with clCreateProgramWithBinary

Task-Parallel Programming Model Hints

Reuse Compilation Results with clCreateProgramWithBinary

If the compilation time for an OpenCL™ program is of concern, consider reusing compilation results. It is typically faster than recreating you program from the source, but you should check for the specific program or device.

To retrieve binaries generated from calls to `clCreateProgramWithSource` and `clBuildProgram`, you can call `clGetProgramInfo` with the `CL_PROGRAM_BINARIES` parameter. For the performance-critical applications that typically precompile kernel code to an Intermediate Representation (IR), you can cache the resulting binaries after the first OpenCL compilation and reuse them on subsequent executions by use of `clCreateProgramWithBinary`. Another way to save intermediate binaries is to use the Intel® SDK for OpenCL™ Applications, as described in the Developer Guide for Intel® SDK for OpenCL™ Applications.

Debugging OpenCL™ Kernels on Linux* OS

The Intel® CPU Runtime for OpenCL™ Applications supports debugging OpenCL™ kernels using a GNU GDB* or a LLDB* Debugger on Linux* OS. It allows for debugging host code and OpenCL kernels in a single debug session.

- Enabling Debugging in OpenCL™ CPU Compiler and Runtime
- Start a Debugging Session
- Conditional Breakpoints on Work Items

See Also

- For more information on the GNU* Project Debugger, see https://www.gnu.org/software/gdb/
- For more information on the LLDB* Debugger, see https://lldb.llvm.org/lldb-gdb.html

Enabling Debugging in OpenCL™ CPU Compiler and Runtime

To enable the debugging mode in the OpenCL™ CPU Compiler and Runtime, specific options should be passed to the build options string parameter in the `clBuildProgram` function:

- `-g` flag enables source-level debug information generation.

  NOTE Passing a `-g` flag means that no optimizations (such as inlining, unrolling, vectorization) are performed, the same as if a `-cl-opt-disable` option was passed.

- `-s /full/path/to/OpenCL/source/file.cl` option specifies the full path to the OpenCL™ source file. If the path includes spaces, the entire path should be enclosed with double or single quotes.

  This option is required in cases where the source of an OpenCL™ kernel is passed to the `clCreateProgramWithSource` function as a string.

  Only one file can be specified with the `-s` option. If the file contains one or more `#include` directives, multiple paths for files with the `-s` option are not needed.

Consider the following example of using the `clBuildProgram` function:

```
err = clBuildProgram(
        g_program,
        0,
        NULL,
        "-g -s \"<path_to_opencl_source_file>\"",
        NULL,
        NULL);
```

NOTE The OpenCL™ kernel code must exist in a text file that is separate from the host code. Debugging OpenCL™ code that appears only in a string embedded in the host application is not supported.

Instead of passing the `-s` option to specify a path to an OpenCL source file, you can use one of the following approaches:

- Use an Intel® SDK for OpenCL™ Applications - Offline Compiler.

  Specifying the `-s` option is not necessary, as the path to an OpenCL source file is defined by the `-input` option:

```
ioc -cmd=build -input=kernel.cl '-bo=-g'
```

- Pass a string with one or more `#include` directives to `clCreateProgramWithSource`.

  In this case, the paths to included files are detected automatically:

```
const char* src = "#include kernel.cl";
  cl_program program = clCreateProgramWithSource(
        m_context, 1, &src, NULL, &error);
```

Start a Debugging Session

The standard GDB* or LLDB* commands are used to debug OpenCL(TM) programs:

```
gdb --args ./host_program
```

Breakpoints and stepping functionality are fully supported.

For more information about GDB* or LLDB* debuggers, see the See Also section below.

1. Start debugging a program:

```
gdb --args ./host_program
```

2. Set a breakpoint in a kernel:

```
(gdb) break kernel.cl:5
     Make breakpoint pending on future shared library load? (y or [n]) y
     Breakpoint 1 (kernel.cl:5) pending.
```

3. Run the host program. The execution stops once the debugger hits the breakpoint in the kernel:

```
   (gdb) run
     Thread 19 "debugger_test_t" hit Breakpoint 1, foo (c=9 '\t') kernel.cl:5
     5          d = d + 2;
```

See Also

- For more information on the GNU* Project Debugger, see https://www.gnu.org/software/gdb/
- For more information on the LLDB* Debugger, see https://lldb.llvm.org/lldb-gdb.html

Conditional Breakpoints on Work Items

According to the OpenCL™ standard, work items are executed concurrently. If a work group contains more than one work item, the debugger stops on a breakpoint in every running work item.

When a kernel is compiled with a `-g` option, the compiler generates three implicit variables that define the current work item by mapping to global ID within NDRange . Each variable corresponds with one dimension in the three-dimensional NDRange:

- `__ocl_dbg_gid0`
- `__ocl_dbg_gid1`
- `__ocl_dbg_gid2`

You can use these variables to set conditional breakpoints on a specific work item (or work items).

Setting Conditional Breakpoints

To set conditional breakpoints, use the native GDB* or LLDB* commands for conditional breakpoints.

Consider the following example of using conditional breakpoints with the `__ocl_dbg_gid0` variable:

1. Place a breakpoint at a work item on dimension zero with global ID value equal to two:

```
(gdb) break kernel.cl:3 if (__ocl_dbg_gid0 == 2)
```

2. Answer yes (`y`) when you see the following message, because at this moment, a kernel code does not exist, and it will be generated only after you run an application:

```
Make breakpoint pending on future shared library load? (y or [n]) y
```

   Expected output for this example:

```
Breakpoint 3 (kernel.cl:3 if (__ocl_dbg_gid0 == 2)) pending.
```

3. Run the application:

```
(gdb) run
```

   If the application successfully stopped on the breakpoint, you will see the following message with a status of the application and a line where the breakpoint was placed:

```
[Switching to Thread 0x7fffcffff700 (LWP 26115)]
Thread 20 "host_program" hit Breakpoint 1, main_kernel (buf_in=0x1834280 "", buf_out=0x186c880
"")
at kernel.cl:3
3           size_t workdim = get_work_dim();
```

4. You can use the `__ocl_dbg_gid0` variable to identify a global ID for a specified NDRange dimension. To print the global ID, use the following command:

```
(gdb) print __ocl_dbg_gid0
```

   Expected output for this example:

```
$0 = 2
```

Performance Debugging with Intel® SDK for OpenCL™ Applications

- Performance Debugging Introduction
- Host-Side Timing
- Profiling Operations Using OpenCL Profiling Events
- Comparing OpenCL™ and Native Code Performance
- Getting Credible Performance Numbers
- Tools for OpenCL™ Development

Performance Debugging Introduction

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 (or for example, average, geometric mean) value for the execution time is usually used for final projections.

Host-Side Timing

The following code snippet is a host-side timing routine around a kernel call (error handling is omitted):

```
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);
```

In this example, host-side timing is implemented using the following functions:

- `clEnqueueNDRangeKernel` puts a kernel to a queue and immediately returns
- `clFinish` explicitly indicates the completion of kernel execution. You can also use `clWaitForEvents`.

Remember to profile operations on memory objects separately.

See Also

Profiling Operations Using OpenCL™ Profiling Events

Profiling Operations Using OpenCL™ Profiling Events

The following code snippet measures kernel execution using the OpenCL™ profiling events (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);
```

Also consider the following:

- Enable the queue for profiling by use of the `CL_QUEUE_PROFILING_ENABLE` property in creation time.
- Explicitly synchronize the operation using `clFinish()` or `clWaitForEvents` because device time counters for the profiled command are associated with the specified event.

Using the profiling operations, you can profile operations on both memory objects and kernels. Refer to section 5.12 of the OpenCL 1.2 Specification for the detailed description of profiling events.

NOTENOTE: The host-side wall-clock time might return different results. For the CPU device, the difference is typically minor.

See Also

Comparing OpenCL™ and Native Code Performance

OpenCL 1.2 Specification at https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf

Comparing OpenCL™ and Native Code Performance

When comparing an OpenCL™ kernel performance on CPU device with native code performance, make sure that both versions of code are as similar as possible. Consider the following guidelines:

- Wrap exactly the same set of operations.
- Do not include program build time in the kernel execution time. You can amortize this step by program precompilation using the `clCreateProgramFromBinary` call.
- Track data transfers costs separately.
- Use data mapping to make data transfers similar to the way data is passed in native code (by use of pointers). Refer to the Mapping Memory Objects (USE_HOST_PTR) section
- Ensure the working set is identical for native and OpenCL code.
- Make the memory access patterns equal (row-wise compared to column-wise).
- Demand the same accuracy. Consider the example for CPU device. `rsqrt(x)` is inherently of the 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 couple of additional Newton-Raphson iterations to match the precision of OpenCL™`rsqrt`.
    - Use `native_rsqrt` in your OpenCL™ kernel, which maps exactly to the `rsqrtps` instruction in the final assembly code.
    - Use the relaxed-math compilation flag to enable similar accuracy for the whole program. Similarly to `rsqrt`, you can use the relaxed versions of `rcp`, `sqrt`, and so on. Refer to the Developer Guide for Intel® SDK for OpenCL™ Applications for the full list of supported functions.

See Also

Getting Credible Performance Numbers

Use Lower Math Precision

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 (or average, geometric mean, and so on) value for the execution time is usually used for final projections.

An alternative to calling kernel several times is using a single "warm-up" run.

The warm-up run might be helpful for kernels with small amount of computations, as 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

NOTENOTE: You need to make your performance conclusions on reproducible data. If warm-up run does not help or execution time still varies, try running large number of iterations and then average the results. For time values that range too much, consider using `geomean`.

Consider the following:

- For bandwidth-limited kernels, operating on the data that does not fit in the last-level cache, the warm-up run does not improve the stability of measurement significantly.
- 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 that the kernel run time is at least 20 milliseconds for CPU device.
- Kernels with smaller run time might provide unreliable data, so increasing the amount of computations artificially gives you important insights into the hotspots. For example, you can add loop in the kernel, or replicate some pieces.

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

See Also

Simple Optimizations of OpenCL™ Code

Tools for OpenCL™ Development

Once you get stable performance numbers, you need to decide what to optimize first.

Use the following tools to optimize your OpenCL™ kernels:

- **Intel® VTune™ Amplifier XE 2018**, which enables you to fine-tune for optimal performance, ensuring the CPU or coprocessor device facilities are fully utilized.
- **Intel® Code Builder for OpenCL™ API**, which enables you to build and analyze your OpenCL kernels. The tool provides full offline OpenCL language compilation.
- **Intel® SDK for OpenCL™ - Offline Compiler command-line 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.
- **Intel® SDK for OpenCL™ Applications – Debugger**, which enables you to debug OpenCL kernels with the GNU Project Debugger (GDB).

For more information on the supported tools, refer to the Developer Guide for Intel® SDK for OpenCL™ Applications .

Coding for the Intel® Architecture Processors

Introduction for OpenCL™ Coding on Intel® Architecture Processors

Each work group is assigned to one thread that loops over all work items within the work group with SIMD. So you have parallelism at the work-group level (vector instructions) and parallelism between work-groups (threading).

Generally, you think in terms of "total work" first, which is "global work size" in OpenCL™ notion. Recall that `global-size = work-group-size*number-of-work-groups`. Thus, understanding the trade-offs between work-group size and number of work-groups is very important for both types of parallelism that we just discussed. In this section certain general recommendations are provided.

Vectorization Basics for Intel® Architecture Processors

Intel® Architecture Processors provide performance acceleration using Single Instruction Multiple Data (SIMD) instruction sets, which include:

- Intel® Streaming SIMD Extensions (Intel® SSE)
- Intel® Advanced Vector Extensions (Intel® AVX) instructions
- Intel® Advanced Vector Extensions 2 (Intel® AVX2) instructions
- Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Foundation instructions, Intel® AVX-512 Conflict Detection instructions, Intel® AVX-512 Doubleword and Quadword instructions, Intel® AVX-512 Byte and Word instructions, and Intel® AVX-512 Vector Length Extensions for Intel® processors

By processing multiple data elements in a single instruction, these ISA extensions enable data parallelism.

When using SIMD instructions, vector registers can store a group of data elements of the same data type, such as `float` or `char`. The number of data elements that fit in one register depends on the microarchitecture and on the data type width: for example, in case CPU supports vector register width 512 bits, each vector (ZMM) register can store sixteen float numbers, sixteen 32-bit integer numbers, and so on.

When using the SPMD technique, the Intel® OpenCL™ implementation can map the work items to the hardware according to one of the following:

- Scalar code, when work-items execute one-by-one
- SIMD elements, when several work-items fit into one register to run simultaneously

The Intel® SDK for OpenCL™ Applications contains an implicit vectorization module, which implements the second method. Depending on the kernel code, this operation might have some limitations. If the vectorization module optimization is disabled, the Intel SDK for OpenCL Applications uses the first method.

See Also

Vectorization: SIMD Processing Within a Work Group

Benefitting From Implicit Vectorization

Vectorization: SIMD Processing Within a Work Group

Intel® SDK for OpenCL™ Applications includes an automatic vectorization module as part of the OpenCL program build process. Depending on the kernel code, this operation might have some limitations. When it is beneficial performance-wise, the module automatically packs adjacent work-items (from dimension zero of the ND-range) and executes them with SIMD instructions.

When using SIMD instructions, vector registers store a group of data elements of the same data type, such as `float` or `int`. The number of data elements that fit in one register depends on the data type width, for example: Intel® Xeon® processor (formerly known Intel® processor code name Skylake) offers vector register width of 512 bits. Each vector register (zmm) can store sixteen float (or alternatively eight double) or sixteen 32-bit integer numbers, and these are the most natural data types to work with Intel Xeon processor. Smaller data types are also processed by 16 elements at a time with some conversions.

A work group is the finest granularity for thread-level parallelism. Different threads pick up different work groups. Thus, per-work-group amount of calculations coupled with right work-group size and the resulting number of work groups available for parallel execution are critical factors in achieving good scalability for Intel Xeon processor.

The vectorization module enables you to benefit from vector units without writing explicit vector code. Also, you do not need `for` loops within kernels to benefit from vectorization. For better results, process a single data element in the kernel and let the vectorization module take care of the rest. To get more performance gains from vectorization, make you OpenCL code as simple as possible.

The vectorization module works best for the kernels that operate on elements of `float` (`double`) or `int` data types. The performance benefit might be lower for the kernels that include a complicated control flow.

The vectorization module packs work items for dimension zero of NDRange. Consider the following code example:

```
___kernel foo(…)
for (int i = 0; i < get_local_size(2); i++)
    for (int j = 0; j < get_local_size(1); j++)
        for (int k = 0; k < get_local_size(0); k++)
                Kernel_Body;
```

After vectorization, the code example of the work group looping over work items appears as follows:

```
___kernel foo(…)
for (int i = 0; i < get_local_size(2); i++)
    for (int j = 0; j < get_local_size(1); j++)
        for (int k = 0; k < get_local_size(0); k+=SIMD_WIDTH)
                VECTORIZED_Kernel_Body;
```

Also note that the dimension zero is the innermost loop and is vectorized. For more information, refer to the Intel® OpenCL™ Implicit Vectorization Module overview at http://llvm.org/devmtg/2011-11/Rotem_IntelOpenCLSDKVectorizer.pdf and Autovectorization in Intel® SDK for OpenCL™ Applications version 1.5.

Benefitting from Implicit Vectorization

Intel® SDK for OpenCL™ Applications includes an implicit vectorization module as part of the program build process. When it is beneficial performance-wise, this module packs several work items and executes them with SIMD instructions. This enables you to benefit from the vector units in the Intel® Architecture processors without writing explicit vector code.

The vectorization module transforms scalar data type operations by adjacent work-items into an equivalent vector operations. When vector operations already exist in the kernel source code, the module scalarizes (breaks down into component operations) and revectorizes them. This improves performance by transforming the memory access pattern of the kernel into a structure of arrays (SOA), which is often more cache-friendly than an array of structures (AOS).

You can find more details in the Intel® OpenCL™ Implicit Vectorization Module overview at http://llvm.org/devmtg/2011-11/Rotem_IntelOpenCLSDKVectorizer.pdf and OpenCL™ Autovectorization in Intel SDK for OpenCL Applications version 1.5.

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

Since the default computation width is four-byte, kernels are vectorized by default. If your kernel uses certain vector, you can specify `__attribute__((vec_type_hint(<typen>)))` with `typen` of any vector type (for example, `float3` or `char4`). This attribute indicates to the vectorization module apply only transformations that are useful for this type.

The performance benefit from the vectorization module might be lower for the kernels that include a complex control flow.

To benefit from vectorization, you do not need the `for` loops within kernels. For best results, let the kernel deal 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 get from vectorization.

Writing the kernel in the plain scalar code is what works best for efficient vectorization. This method of coding avoids many disadvantages potentially associated with explicit (manual) vectorization described in the Using Vector Data Types section.

See Also

Vectorizer Knobs

Vectorizer Knobs

Several environment variables and one attribute extension are related to vectorizer.

Environment Variables

This variable can be set to False and True respectively. Notice that just like any other environment variables this one affects the behavior of the vectorizer of the entire system (or shell instances) until variable gets unset explicitly (or shell(s) terminates).

This variable affects code generation for CPU OpenCL device only. It effectively sets the vectorization "width" (when `CL_CONFIG_USE_VECTORIZER = True`):

- `CL_CONFIG_CPU_VECTORIZER_MODE = 0` (default). The compiler makes heuristic decisions whether to vectorize each kernel, and if so which vector width to use.
- `CL_CONFIG_CPU_VECTORIZER_MODE = 1`. No vectorization by compiler. Explicit vector data types in kernels are left intact. This mode is the same as `CL_CONFIG_USE_VECTORIZER = False`.
- `CL_CONFIG_CPU_VECTORIZER_MODE = 4`. Disables heuristic and vectorizes to the width of 4.
- `CL_CONFIG_CPU_VECTORIZER_MODE = 8`. Disables heuristic and vectorizes to the width of 8.
- `CL_CONFIG_CPU_VECTORIZER_MODE = 16`. Disables heuristic and vectorizes to the width of 16.

Attribute Extensions

The goal of this extension is to allow programmers to specify vector length the kernel should be vectorized to. This information may be used by the compiler to generate more optimal code.

For a device that supports the extension, the function `clGetDeviceInfo` with the parameter `CL_DEVICE_EXTENSION` returns a space separated list of extensions names that contains `cl_intel_vec_len_hint`.

Use the OpenCL C optional attribute qualifier `__attribute__((intel_vec_len_hint(<int>)))`, where `<int>` is a vector length that the kernel should be vectorized to.

You can set one the following value to the variable:

- `0` - the compiler makes heuristic decisions whether to vectorize each kernel, and if so which vector length to use.
- `1` - no vectorization by compiler. Explicit vector data types in kernels are left intact.
- `4` - disables heuristic and vectorizes to the length of 4.
- `8` - disables heuristic and vectorizes to the length of 8.
- `16` - disables heuristic and vectorizes to the length of 16.

NOTE Note

NOTE If the work group size is not evenly divisible by the specified vector length hint, loop remainder might not be executed in vector code iterations.

NOTE Note

NOTE If you specify simultaneously the `intel_vec_len_hint` and `vec_type_hint` attributes, the compiler ignores `vec_type_hint` attribute.

The examples below illustrate valid and invalid uses of the extension:

- Vectorizing to the length of 8:

```
__attribute__((intel_vec_len_hint(8)))
__kernel void kr1(…) {
…
}
```

- Specifying both `intel_vec_len_hint` and `vec_type_hint`. In the example, the compiler ignores `vec_type_hint` and vectorize to the length of 4.

```
__attribute__((intel_vec_len_hint(4)))
__attribute__((vec_type_hint (float8)))
__kernel void kr2(…) {
…
}
```

See Also

Developer Guide for Intel® SDK for OpenCL™ Applications

Targeting a Different CPU Architecture

This variable generates code exclusively for a given target CPU architecture.

NOTE`CL_CONFIG_CPU_TARGET_ARCH` allows only lowering the instruction set level supported by CPU.

By default, it is set to `Autodetect`.

Allowed values are:

- `CL_CONFIG_CPU_TARGET_ARCH = skx`. Generates code for processors that support Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Foundation instructions, Intel® AVX-512 Conflict Detection instructions, Intel® AVX-512 Doubleword and Quadword instructions, Intel® AVX-512 Byte and Word instructions and Intel® AVX-512 Vector Length Extensions for Intel® processors, and the instructions enabled with `core-avx2`.
- `CL_CONFIG_CPU_TARGET_ARCH = core-avx2`. Generates code for processors that support Intel® Advanced Vector Extensions 2 (Intel® AVX2), Intel® AVX, SSE4.2 SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions.
- `CL_CONFIG_CPU_TARGET_ARCH = corei7-avx`. Generates code for processors that support Intel® Advanced Vector Extensions (Intel® AVX), Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions.
- `CL_CONFIG_CPU_TARGET_ARCH = corei7`. Generates code for processors that support Intel® SSE4.2 Efficient Accelerated String and Text Processing instructions. May also generate code for Intel® SSE4 Vectorizing Compiler and Media Accelerator, Intel® SSE3, SSE2, SSE, and SSSE3 instructions.

NOTE Some kernels are not possible to be vectorized, so vectorizer would not touch them regardless of the mode. Also be careful with manual overriding the compiler heuristic, build process would fail if target hardware does not support the specific vectorization width. Inspect the compiler output in the offline compiler tool (described in the Developer Guide) on the messages related to vectorization.

Using Vector Data Types

To maximize use of vector CPUs, consider using vector data types in your kernel code as a more involved performance alternative to the automatic (compiler-aided) vectorization described in the Benefitting from Implicit Vectorization section. This technique enables you to map vector data types directly to the hardware vector registers. Thus, the used data types should match the width of the underlying SIMD instructions.

Consider the following recommendations:

- Starting the 2nd Generation Intel® Core™ processors with Intel® Advanced Vector Extension (Intel® AVX) support, use data types such as `float8` or `double4`, so you bind code to the specific register width of the underlying hardware. This method provides maximum performance on a specific platform. However, performance on other platforms and generations of Intel® Core™ processors might be less than optimal.
- Use wider data types, such as `float16`, to transparently cover many SIMD hardware register widths. However, using types wider than the underlying hardware is similar to loop unrolling. This method might improve performance in some cases, but also increases register pressure. Consider using `uchar16` data type to process four pixels simultaneously when operating on pixels with eight bits per component.
- With vector data types, each work item processes `N` elements. Make sure the size of a grid, which is the number of work-items required to process the same dataset, does not exceed the N value.

NOTENOTE: The `int8` data type improves performance only starting the 4th Generation Intel® Core™ processors.

Using vector data types, you plan the vector-level parallelism yourself instead of relying on the implicit vectorization module. See the Benefitting from Implicit Vectorization section for more information.

This approach is useful in the following scenarios:

- You are porting the code that originally used the following instructions:
  - Intel® Streaming SIMD Extensions (Intel® SSE)
  - Intel® Advanced Vector Extensions (Intel® AVX)
  - Intel® Advanced Vector Extensions 2 (Intel® AVX2)
  - Intel® Advanced Vector Extensions 512 (Intel® AVX-512)
- You want to benefit from hand-tuned vectorization of your code.

The following example demonstrates the multiplication kernel that targets the 256-bit vector units of the 2nd Generation Intel Core processors and higher:

```
__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 eight elements together.

The attribute added before the kernel, signals the compiler, or the implementation that this kernel has an optimized vectorized form, so the implicit vectorization module does not operate on it. Use `vec_type_hint` to hint compiler that your kernel already processes data using mostly vector types. For more details on this attribute, see the section 6.7.2 of the OpenCL™ 1.2 specification at https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf.

See Also

Writing Kernels to Directly Target the Intel Architecture Vector Processors

OpenCL 1.2 Specification at https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf

Writing Kernels to Directly Target the Intel® Architecture Processors

Using the OpenCL™ vector data types is a straightforward way to directly utilize the Intel® Architecture vector instruction set. See the Using Vector Data Types section for more information. Consider the following code 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, an OpenCL kernel that uses `float4` data type, transparently benefits from Intel® Advanced Vector Extensions (Intel® AVX) if the compiler promotes `float4` to `float8`. The vectorization module can pack work items automatically, though it might be sometimes less efficient than manual packing.

If the native size for your kernel requires less than 128 bits and you want to benefit from the explicit vectorization, consider packing work items together manually.

For example, your kernel uses the `float2` vector type. It receives (`x`, `y`) float coordinates, and shifts them by (`dx`, `dy`):

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

To increase the kernel performance, you can manually pack 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 job as a work item in the previous kernel. Consequently, they require only one fourth of invocations, reducing the run-time overheads. However, when you use manual packing, you must also change the host code accordingly.

For vectors of 32-bit data types, for example, `int4`, `int8`, `float4`, and `float8` data types use explicit vectorization to improve the performance. Other data types (for example, `char3`) may cause a behind-the-scene upcast of the input data, which has negative impact on performance.

For best performance for a given data type, the vector width should match the underlying SIMD width. This value differs for different architectures. For example, consider querying the recommended vector width using `clGetDeviceInfo` with `CL_DEVICE_PREFERRED_VECTOR_WIDTH_INT` parameter. You get vector width of four for the 2nd Generation Intel® Core™ processors, but vector width of eight for higher versions of processors. Use `int8` so that vector width fits both architectures. Similarly for floating point data types, you can use `float8` data to cover many potential architectures.

NOTENOTE: Using scalar data types such as `int` or `float` is often the most “scalable” way to help the compiler do right vectorization job for the specific SIMD architecture underneath.

You can target to a specific Intel Architecture processor using a conditional code with an OpenCL™ C predefined macro `__INTEL_OPENCL_CPU_<CPUSIGN>`.

The macro tunes the kernel for a specific CPU device microarchitecture. `<CPUSIGN>` is the CPU signature of a device.

You can specify one of the following values for this macro:

- `__INTEL_OPENCL_CPU_SKL__` - Intel® microarchitecture code name Skylake
- `__INTEL_OPENCL_CPU_SKX__` - Intel® microarchitecture code name Skylake on Intel Xeon® processor family
- `__INTEL_OPENCL_CPU_BDW__` - Intel® microarchitecture code name Broadwell
- `__INTEL_OPENCL_CPU_BDW_XEON__` - Intel® microarchitecture code name Broadwell on Intel Xeon® processor family
- `__INTEL_OPENCL_CPU_HSW__` - Intel® microarchitecture code name Haswell
- `__INTEL_OPENCL_CPU_HSW_XEON__` - Intel® microarchitecture code name Haswell on Intel Xeon® processor family
- `__INTEL_OPENCL_CPU_IVB__` - Intel® microarchitecture code name Ivy Bridge
- `__INTEL_OPENCL_CPU_IVB_XEON__` - Intel® microarchitecture code name Ivy Bridge on Intel Xeon® processor family
- `__INTEL_OPENCL_CPU_SNB__` - Intel® microarchitecture code name Sandy Bridge
- `__INTEL_OPENCL_CPU_SNB_XEON__` - Intel® microarchitecture code name Sandy Bridge on Intel Xeon® processor family
- `__INTEL_OPENCL_CPU_WST__` - Intel® microarchitecture code name Westmere
- `__INTEL_OPENCL_CPU_WST_XEON__` - Intel® microarchitecture code name Westmere on Intel Xeon® processor family
- `__INTEL_OPENCL_CPU_UNKNOWN__` - Unknown microarchitecture

To tune performance for your target CPU, you can use this macro with `intel_vec_len_hint` extension. For example:

```
// Kernel side.

// Force vectorization with to 8 on BDW.
// Runtime defines a macro corresponding to the device CPU signature.
#ifdef __INTEL_OPENCL_CPU_BDW__
__attribute__((intel_vec_len_hint(8)))
#endif //BDW
__kernel void memcpy1(__global float* src, __global float* dst)
{
    size_t gid = get_global_id(0);
    dst[gid] = src[gid];
}
```

For more information about `intel_vec_len_hint` attribute extension, refer to Vectorizer Knobs.

See Also

OpenCL 1.2 Specification at https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf

Work-Group Size Considerations

The recommended work-group size for kernels is multiple of 4, 8, or 16, depending on Single Instruction Multiple Data (SIMD) width for the `float` and `int` data type supported by CPU. The automatic vectorization module packs the work-items into SIMD packets of 4/8/16 items (for double as well) and processed the rest ("tail") of the work group in a scalar way. In other words, a work-group with the size of `2*SIMD_WIDTH` executes faster than a work-group with the size of `2*SIMD_WIDTH-1`.

For example, a work group of 64 elements is assigned to one hardware thread. The thread iterates over work-items in a loop of 4 iterations with 16-wide vector instructions within each iteration. In some cases, the compiler may decide to loop (unroll) by 32 elements instead to expose more instruction-level parallelism.

It is recommended to let the OpenCL™ implementation automatically determine the optimal work-group size for a kernel: pass `NULL` for a pointer to the work-group size when calling `clEnqueueNDRangeKernel`.

If you want to experiment with work-group size, you need to consider the following:

- To get best performance from using the vectorization module (see the Benefitting from Implicit Vectorization section), the work-group size must be larger or a multiple of 4, 8, or 16 depending on the SIMD width supported by CPU otherwise case the runtime can make a wrong guess of using the work-groups size of one, which results in running the scalar code for the kernel.
- To accommodate multiple architectures, query the device for the `CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE` parameter by calling to `clGetKernelWorkGroupInfo` and set the work-group size accordingly.
- To reduce the overhead of maintaining a workgroup, you should create work-groups that are as large as possible, which means 64 and more work-items. One upper bound is the size of the accessed data set as it is better not to exceed the size of the L1 cache in a single work group. Also there should be sufficient number of work-groups, see the Work-Group Level Parallelism section for more information.
- If your kernel code contains the barrier instruction, 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 is. The reason is that a barrier also issues copy instructions for the total amount of private and local memory used by all work-items in the work-group in the work-group since the state of each work-item that arrived at the barrier is saved before proceeding with another work-item. Make the work-group size be multiple of 4, 8, or 16, otherwise the scalar version of the resulted code might execute.

Threading: Achieving Work-Group Level Parallelism

Since work-groups are independent, they can execute concurrently on different hardware threads. So the number of work-groups should be not less than the number of logical cores. A larger number of work-groups results in more flexibility in scheduling, at the cost of task-switching overhead.

Also notice that in the opposite case, when the number of work-groups is relatively small, in compare to, for example the value of `CL_DEVICE_MAX_COMPUTE_UNITS`, then even a small change in the work-groups amount can result in a significant performance change.

For example, if you run a number of work-groups that equals to `CL_DEVICE_MAX_COMPUTE_UNITS`, then each compute unit process exactly one work-group. So in ideal conditions all threads finish at the same time. Now consider the case, when work-group size is changed, so that `CL_DEVICE_MAX_COMPUTE_UNITS+1` work-groups are executed instead. In such case, one thread does two times more job than the others, which might double the overall execution time. Some inherent threads divergence might hide the effect. The negative effect of "outstanding" work-groups is less and less pronounced as the number of work-groups grows, since imbalance is decreasing at a same pace.

Notice that multiple cores of a CPU as well as multiple CPUs (in a multi-socket machine) constitute a single OpenCL™ device. Separate cores are compute units. The device fission extension enables you to control compute unit use within a compute device. For more information on the device fission, refer to the OpenCL™ Device Fission for CPU Performance.

For the best performance and parallelism between work-groups, ensure that execution of a work-group takes at least 100,000 clocks. A smaller value increases the proportion of switching overhead compared to actual work.

Efficient Data Layout

The vectorization module transforms scalar data type operations on adjacent work-items into an equivalent vector operation. If vector operations already exist in the kernel source code, the module scalarizes (breaks into component operations) and revectorizes them. Such operation improves performance by transforming the memory access pattern of the kernel into a structure of arrays (SOA), which is often more cache-friendly than an array of structures (AOS).

This transformation comes with a certain cost. Organizing the input data in SOA reduces the transpose penalty. For example, the following code example suffers from transpose overhead:

```
__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 next piece of code does 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];
}
```

To make the vectorization the most efficient, the sequential work items should refer to sequential memory locations. Otherwise, data gathering required for processing in SIMD might be expensive performance-wise. For example:

```
int tid  = get_global_id(0);
output[tid] = inx[tid]; //sequential access (with respect to adjacent work-items)
output[tid] = inx[2*tid]; //non-sequential access (triggers data gathering)
```

There is an alternative to AOS that generally preserves the expressiveness of AOS and efficiency of SOA. It is sometimes referenced as strided (or “stripped)” SOA, or even AOSSOA. Consider the code example below:

```
struct AOSSOA
{
float [16] x;
float [16] y;
float [16] z;
};
```

Using the Blocking Technique

An important class of algorithmic changes involves changing the access pattern so that working set fits in cache. By organizing data memory accesses, you can load the cache with a small subset of a much larger data set. The idea is then to work on this block of data in cache. By using/reusing this data in cache we reduce the need to go to memory (memory bandwidth pressure).

Blocking is an optimization technique that can help to avoid memory bandwidth bottlenecks. Blocking enables you to exploit the inherent data reuse available in the application by ensuring that the data remains in cache across multiple uses. You can use the blocking technique on one-, two- or three-dimension spatial data structures. Some iterative applications can benefit from blocking over multiple iterations, which is called temporal blocking, and further mitigate bandwidth bottlenecks, which requires merging some kernels. In terms of code change, blocking typically involves loop splitting. In most application codes, blocking is best performed by parameterization and tuning of the block-factors.

For instance, the code snippet below shows an example of blocking with two loops (`i1` and `i2`) iterating over entire data set. The original code streams through the entire set in the inner loop, and must load the `data[i2]` value from memory in each iteration. Assuming `NUM` is large, and compute is not arithmetic intensive, we have no reuse in cache so this application is memory-bandwidth bound.

The original code without applying the blocking technique appears as follows:

```
for (i1 = 0; i1 < NUM; i1 ++){
        for (i2=0; i2 < NUM; i2++) {
                  OUT[i1] += compute(data[i1], data[i2]);
        }
}
```

The blocked code below is obtained by splitting the `i2` loop into an outer loop iterating over bodies in multiple of `BLOCK`, and an inner `i22` loop, iterating over elements within the block; and interleaving the `i1` and `i2` loops. This code reuses a set of `BLOCKi2` values across multiple iterations of the `i1` loop. In case you choose `BLOCK` so that it makes the set of values fit in the L2 cache, the memory traffic is reduced by a factor of the `BLOCK`.

The modified code with use of one-dimensional blocking appears as follows:

```
for (i2 = 0; i2 < NUM; i2 += BLOCK) {
   for (i1=0; i1 < NUM; i1 ++) {
         for (i22=0; i22 < BLOCK; i22 ++) {
             OUT[i1] += compute(data[i1], data[i2 + i22]);
                }
        }
}
```

For the more detailed example of tiling, refer to the General Matrix Multiply Sample .

Intel® Turbo Boost Technology Support

Intel® Turbo Boost Technology applies to CPU cores. The processor must run within the thermal constraints of the system. Thus CPU core might boost and throttle the frequency according to the current performance and power-saving balance.

Global Memory Size

OpenCL™ "global" memory is allocated from system (host) memory for CPU. The amount of available memory depends on the amount of computer system memory and the operating system number of bits. For example, a system with 4GB of RAM running on a 32-bit OS usually has less than 3GB available for system memory. This impacts the amount of global memory available for the CPU device. Use the `clGetDeviceInfo(…,CL_DEVICE_GLOBAL_MEM_SIZE)` query to get information on the exact available amount. If you fail to allocate resources required by the OpenCL™ implementation on the device, you receive the `CL_OUT_OF_RESOURCES` error.

Global memory performance depends on the frequency of DDR memory.

Since global memory is shared between CPU and the system host, it is important to use mapping for memory objects. See the Map Memory Objects (USE_HOST_PTR) section for more information.