## Contents

**Altera SDK for OpenCL Optimization Guide**.................................1-1

About Altera SDK for OpenCL Optimization Guide.................................1-1
Introduction..............................................................................................1-1
  FPGA Overview......................................................................................1-1
  Pipelines..............................................................................................1-2
Good Design Practices...............................................................................1-4
  Avoid Pointer Aliasing......................................................................1-5
  Avoid Expensive Functions.................................................................1-5
  Avoid Work-Item ID-Dependent Backward Branching..............................1-6
  Aligned Memory Allocation.................................................................1-7
  Ensure 4-Byte Alignment for All Data Structures.....................................1-7
  Maintain Similar Structures for Vector Type Elements.............................1-8
Optimization of Data Processing Efficiency........................................1-8
  Specify a Maximum Work-Group Size or a Required Work-Group Size.........1-9
  Loop Unrolling...................................................................................1-10
  Resource Sharing..............................................................................1-12
  Kernel Vectorization.........................................................................1-13
  Multiple Compute Units.....................................................................1-15
  Combination of Compute Unit Replication and Kernel SIMD Vectorization..1-17
  Resource-Driven Optimization...........................................................1-18
  Floating-Point Operations.................................................................1-20
Optimization of Memory Access Efficiency..........................................1-23
  General Guidelines on Optimizing Memory Accesses............................1-24
  Optimize Global Memory Accesses......................................................1-24
  Perform Kernel Computations Using Constant, Local or Private Memory......1-28
  Single Work-Item Execution...............................................................1-30
Document Revision History....................................................................1-36
About Altera SDK for OpenCL Optimization Guide

The Altera Software Development Kit (SDK) for OpenCL Optimization Guide provides guidance on optimizing your OpenCL applications for Altera field programmable gate arrays (FPGAs).

Audience

The Altera SDK for OpenCL (AOCL) Optimization Guide assumes that you are familiar with OpenCL concepts and application programming interfaces (APIs), as described in the OpenCL Specification version 1.0 by the Khronos Group™. This document also assumes that you have experience in creating OpenCL applications, and are familiar with the contents of the OpenCL Specification.

Related Information

- **OpenCL Specification version 1.0**
  Refer to the OpenCL Specification version 1.0 for detailed information on the OpenCL API and programming language.

- **OpenCL Reference Pages**
  Refer to the OpenCL Reference Pages for more information on the OpenCL Specification version 1.0.

Introduction

To achieve the highest performance of your OpenCL application for FPGAs, you must familiarize yourself with some details of the underlying hardware. In addition, you must understand the compiler optimizations that convert and map your OpenCL application to FPGAs.

FPGA Overview

FPGAs are integrated circuits that you can configure repeatedly to perform an infinite number of functions. With FPGAs, low-level operations like bit masking, shifting, and addition are all configurable. Also, you can assemble these operations in any order. To implement computation pipelines, FPGAs integrate combinations

---

(1) The Altera SDK for OpenCL is based on a published Khronos Specification, and has passed the Khronos Conformance Testing Process. Current conformance status can be found at [www.khronos.org/conformance](http://www.khronos.org/conformance).
(2) OpenCL and the OpenCL logo are trademarks of Apple Inc. and used by permission of the Khronos Group™.

© 2013 Altera Corporation. All rights reserved. ALTERA, ARRIA, CYCLONE, HARDCOPY, MAX, MEGACORE, NIOS, QUARTUS and STRATIX words and logos are trademarks of Altera Corporation and registered in the U.S. Patent and Trademark Office and in other countries. All other words and logos identified as trademarks or service marks are the property of their respective holders as described at [www.altera.com/common/legal.html](http://www.altera.com/common/legal.html). Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
of lookup tables (LUTs), registers, on-chip memories, and arithmetic hardware (for example, digital signal processor (DSP) blocks) through a network of reconfigurable connections. As a result, FPGAs achieve a high level of programmability. LUTs are responsible for implementing various logic functions. For example, reprogramming a LUT can change an operation from a bit-wise AND logic function to a bit-wise XOR logic function.

The key benefit of using FPGAs for algorithm acceleration is that they support wide and heterogeneous pipeline implementations. Each pipeline implemented in the FPGA fabric can be wide and unique. This characteristic is in contrast to many different types of processing units such as symmetric multiprocessors, DSPs, and graphics processing units (GPUs). In these types of devices, parallelism is achieved by replicating the same generic computation hardware multiple times. In FPGAs, however, you can achieve parallelism by duplicating only the logic that your algorithm exercises.

A processor implements an instruction set that limits the amount of work it can perform each clock cycle. For example, most processors do not have a dedicated instruction that can execute the following C code:

\[ E = (((A + B) \oplus C) \& D) \gg 2; \]

Without a dedicated instruction for this C code example, a CPU, DSP, or GPU must execute multiple instructions to perform the operation. In contrast, you may think of an FPGA as a hardware platform that can implement any instruction set that your software algorithm requires. You can configure an FPGA to perform a sequence of operations that implements the code example above in a single clock cycle. An FPGA implementation connects specialized addition hardware with a LUT that performs the bit-wise XOR and AND operations. The device then leverages its programmable connections to perform a right shift by two bits without consuming any hardware resources. The result of this operation then becomes a part of subsequent operations to form complex pipelines.

### Pipelines

The designs of microprocessors, DSPs, hardware accelerators, and other high performance implementations of digital hardware often contain pipeline architectures. In a pipelined architecture, input data passes through a sequence of stages. Each stage performs an operation, such as calculation, memory operation or decoding, that contributes to the final result.

For example, the diagram below represents the following example code fragment as a multistage pipeline:

```c
for(i = 0; i < 1024; i++)
{
    y[i] = (a[i] + b[i] + c[i] + d[i] + e[i] + f[i] + g[i] + h[i]) >> 3;
}
```

![Figure 1-1: Example Multistage Pipeline Diagram](image)
With a pipeline architecture, each arithmetic operation passes into the pipeline one at a time. Therefore, as shown in the diagram above, a saturated pipeline consists of eight stages that calculate the arithmetic operations simultaneously and in parallel. In addition, because of the large number of loop iterations, the pipeline stages continue to perform these arithmetic instructions concurrently for each subsequent loop iteration.

**AOCL Pipeline Approach**

The AOCL pipeline does not have a set of predefined pipeline stages or instruction set. As a result, it can accommodate for the highly configurable nature of FPGAs.

Consider the following OpenCL code fragment:

```c
size_t index = get_global_id(0);
C[index] = (A[index] >> 5) + B[index];
F[index] = (D[index] - E[index]) << 3;
G[index] = C[index] + F[index];
```

You can configure an FPGA to instantiate a complex pipeline structure that executes the entire code in one iteration. In this case, the AOCL implements the code as two independent pipelined entities that feed into a pipelined adder, as shown in the figure below.

**Figure 1-2: Example of the AOCL Pipeline Approach**

The Altera Offline Compiler (AOC) provides a custom pipeline structure that speeds up computation by allowing operations within a large number of work-items to occur concurrently. The AOC can create a custom pipeline that calculates the values for variables \( C, F \) and \( G \) every clock cycle, as shown below. After a ramp-up phase, the pipeline sustains a throughput of one work-item per cycle.
A traditional processor has a limited set of shared registers. Eventually, a processor must write the stored data out to memory to allow more data to occupy the registers. The AOC keeps data "live" by generating enough registers to store the data for all the active work-items within the pipeline. The following code example and figure illustrate a live variable $C$ in the OpenCL pipeline:

```c
size_t index = get_global_id(0);
C = A[index] + B[index];
E[index] = C - D[index];
```

**Figure 1-4: An FPGA Pipeline with a Live Variable C**

Good Design Practices

With the AOC technology, you do not need to change your kernel to fit it optimally into a fixed hardware architecture. Instead, the AOC customizes the hardware architecture automatically to accommodate your kernel requirements.

In general, you should optimize a kernel that targets a single compute unit first. Once you optimize this compute unit, you can increase the performance by scaling the hardware to fill the remainder of the FPGA. The hardware footprint of the kernel correlates with the time it takes for hardware compilation. Therefore,
the more optimizations you can perform with a smaller footprint (that is, a single compute unit), the more hardware compilations you can perform in a given amount of time.

In addition to data processing and memory access optimizations, consider implementing the following design practices, if applicable, when you create your kernels.

**Avoid Pointer Aliasing**

Insert the `restrict` keyword in pointer arguments whenever possible. Including the `restrict` keyword in pointer arguments prevents the AOC from creating unnecessary memory dependencies between non-conflicting load and store operations.

The `restrict` keyword informs the AOC that the pointer does not alias other pointers. For example, if your kernel has two pointers to global memory, `A` and `B`, that never overlap each other, declare the kernel in the following manner:

```c
__kernel void myKernel(__global int * restrict A, __global int * restrict B)
```

**Warning:** Inserting the `restrict` keyword on a pointer that aliases other pointers might result in functional incorrectness.

**Avoid Expensive Functions**

Some functions are expensive to implement in FPGAs. Expensive functions might decrease kernel performance or require a large amount of hardware to implement.

The following functions are expensive:

- Integer division and modulo (remainder) operators
- Most floating-point operators except addition, multiplication, absolution, and comparison

  **Note:** For more information on optimizing floating-point operations, refer to the *Floating-Point Operations* section.

- Atomic functions

In contrast, inexpensive functions have minimal effects on kernel performance, and their implementation consumes minimal hardware.

The following functions are inexpensive:

- Binary logic operations such as AND, NAND, OR, NOR, XOR, and XNOR
- Logical operations with one constant argument
- Shift by constant
- Integer multiplication and division by a constant that is to the power of two

If an expensive function produces a new piece of data for every work-item in a work-group, it is beneficial to code it in a kernel. On the contrary, the code example below shows a case of an expensive floating-point
operation (division) executed by every work-item in the NDRange. The result of this calculation is always the same.

```c
__kernel (__global const float * restrict a,
    __global float * restrict b,
    const float c, const float d)
{
    size_t gid = get_global_id(0);
    //inefficient since each work-item must calculate c divided by d
    b[gid] = a[gid] * (c / d);
}
```

You can avoid this redundant and hardware resource intensive operation by modifying the code example in the following manner. Perform the calculation in the host application and then pass the result to the kernel as an argument, as shown below, to be used by all work-items in the NDRange.

```c
__kernel (__global const float * restrict a,
    __global float * restrict b,
    const float c_divided_by_d)
{
    size_t gid = get_global_id(0);
    /*host calculates c divided by d once and passes it into
    kernel to avoid redundant expensive calculations*/
    b[gid] = a[gid] * c_divided_by_d;
}
```

The AOC consolidates operations that are not work-item-dependent across the entire NDRange into a single operation, and then shares the result across all work-items. In the first code example, the AOC creates a single divider block shared by all work-items because division of $c$ by $d$ remains constant across all work-items. This optimization helps minimize the amount of redundant hardware; however, the implementation of an integer division requires a significant amount of hardware resources. Therefore, it is beneficial to off-load the division operation to the host processor and then pass the result as an argument to the kernel to conserve hardware resources.

**Related Information**

Floating-Point Operations on page 1-20

**Avoid Work-Item ID-Dependent Backward Branching**

The AOC collapses conditional statements into single bits that indicate whether a particular functional unit becomes active. The AOC completely eliminates simple control flows that do not involve looping structures, resulting in a flat control structure and more efficient hardware usage. The AOC compiles efficiently kernels that include forward branches such as `if-then-else` statements. However, you should avoid including any work-item ID-dependent backward branching (that is, branching that occurs in a loop) in your kernel because it degrades performance.
For example, the code fragment below illustrates branching that involves work-item ID such as `get_global_id` or `get_local_id`:

```c
for (size_t i = 0; i < get_global_id(0); i++)
{
    // statements
}
```

**Aligned Memory Allocation**

You should allocate host-side buffers to be at least 64-byte aligned. This allows direct memory access (DMA) transfers to occur to and from the FPGA, which improves buffer transfer efficiency.

To set up aligned memory allocations, add the following source code to your host program:

- For Windows:
  ```c
  #define AOCL_ALIGNMENT 64
  #include <malloc.h>
  void *ptr = _aligned_malloc (size, AOCL_ALIGNMENT);
  ```
  To free up an aligned memory block, include the function call `aligned_free(ptr);`

- For Linux:
  ```c
  #define AOCL_ALIGNMENT 64
  #include <stdlib.h>
  void *ptr = NULL;
  posix_memalign (&ptr, AOCL_ALIGNMENT, size);
  ```
  To free up an aligned memory block, include the function call `free(ptr);`

**Ensure 4-Byte Alignment for All Data Structures**

Alignments smaller than 4 bytes result in larger and slower hardware.

Consider `struct Pixel`, shown in the code example below:

```c
typedef struct
{
    char r, g, b, alpha;
} Pixel; // 1-byte aligned
```

Pixel is only 1-byte aligned. To force 4-byte alignment, create a union of the 1-byte-aligned structure and an integer, as shown in the code example below:

```c
typedef struct
{
    char r, g, b, alpha;
} Pixel_s;
```
typedef union
{
    Pixel_s p;
    int not_used;
}
Pixel; // 4-byte aligned

Maintain Similar Structures for Vector Type Elements

If you update one element of a vector type, update all the elements of the vector.

The code example below illustrates a scenario where you should update a vector element:

```c
__kernel void update (__global const float4 * restrict in,
                      __global const float4 * restrict out)
{
    size_t gid = get_global_id(0);
    out[gid].x = process(in[gid].x);
    out[gid].y = process(in[gid].y);
    out[gid].z = process(in[gid].z);
    out[gid].w = 0; //Update w even if that variable is not required.
}
```

Optimization of Data Processing Efficiency

Consider the following kernel code:

```c
__kernel void sum (__global const float * restrict a,
                   __global const float * restrict b,
                   __global float * restrict answer)
{
    size_t gid = get_global_id(0);
    answer[gid] = a[gid] + b[gid];
}
```

This kernel adds arrays a and b, one element at a time. Each work-item is responsible for adding two elements, one from each array, and storing the sum into the array answer. Without optimization, the kernel performs one addition per work-item.

To maximize the performance of your OpenCL kernel, consider implementing the applicable optimization techniques to improve data processing efficiency.

1. **Specify a Maximum Work-Group Size or a Required Work-Group Size** on page 1-9
   Specify the max_work_group_size or reqd_work_group_size attribute for your kernels whenever possible. These attributes allow the AOC to perform aggressive optimizations to match the kernel to hardware resources without any excess logic.

2. **Loop Unrolling** on page 1-10
   If your OpenCL kernel contains loop iterations, you can increase performance by unrolling the loop.
3. **Resource Sharing** on page 1-12
   Share infrequently-used hardware to free up resources for implementing more single instruction multiple data (SIMD) vector lanes and compute units.

4. **Kernel Vectorization** on page 1-13
   Kernel vectorization allows multiple work-items to execute in SIMD fashion.

5. **Multiple Compute Units** on page 1-15
   To achieve higher throughput, the AOC can generate multiple compute units for each kernel.

6. **Combination of Compute Unit Replication and Kernel SIMD Vectorization** on page 1-17
   Sometimes, your replicated or vectorized kernel might not fit in the FPGA. In that case, you can modify the kernel by both replicating the compute unit and vectorizing the kernel.

7. **Resource-Driven Optimization** on page 1-18
   To optimize throughput, you can enable the resource-driven optimizer to direct the AOC to analyze automatically the effects of combining various values of kernel attributes.

8. **Floating-Point Operations** on page 1-20
   For floating-point operations, you have the option to direct the AOC to perform optimizations that create more efficient pipeline structures in hardware and reduce the overall hardware usage.

---

### Specify a Maximum Work-Group Size or a Required Work-Group Size

Specify the `max_work_group_size` or `reqd_work_group_size` attribute for your kernels whenever possible. These attributes allow the AOC to perform aggressive optimizations to match the kernel to hardware resources without any excess logic.

The AOC assumes a default work-group size for your kernel depending on certain constraints imposed during compilation time and runtime.

The AOC imposes the following constraints at compilation time:

- If you specify a value for the `reqd_work_group_size` attribute, the work-group size must match this value.
- If you specify a value for the `max_work_group_size` attribute, the work-group size must not exceed this value.
- If you do not specify values for `reqd_work_group_size` and `max_work_group_size`, and the kernel contains a barrier, the AOC defaults to a maximum work-group size of 256 work-items.
- If you do not specify values for both attributes, and the kernel contains no barrier, the AOC does not impose any constraint on the work-group size at compilation time.

**Tip:** Use the `CL_KERNEL_WORK_GROUP_SIZE` and `CL_KERNEL_COMPILE_WORK_GROUP_SIZE` queries to the `clGetKernelWorkGroupInfo` API call to determine the work-group size constraints the AOC imposes on a particular kernel at compilation time.

The OpenCL standard imposes the following constraints at runtime:

- The work-group size in each dimension must divide evenly into the requested NDRange size in each dimension.
- The work-group size must not exceed the device constraints specified by the `CL_DEVICE_MAX_WORK_GROUP_SIZE` and `CL_DEVICE_MAX_WORK_ITEM_SIZES` queries to the `clGetDeviceInfo` API call.
If the work-group size you specify for a requested NDRange kernel execution does not satisfy all of the constraints listed above, the `clEnqueueNDRangeKernel` API call fails with the error `CL_INVALID_WORK_GROUP_SIZE`.

Caution: If you do not specify values for both the `reqd_work_group_size` and `max_work_group_size` attributes, the runtime determines a default work-group size as follows:

- If the kernel contains a barrier or refers to the local work-item ID, or if you use the `clGetKernelWorkGroupInfo` and `clGetDeviceInfo` API calls in your host code to query the work-group size, the runtime defaults the work-group size to one work-item.

Important: If your kernel uses memory barriers, local memory, or local work-item IDs, or if your host program queries the work-group size, specify an explicit work-group size when you queue an NDRange kernel (that is, not a single work-item kernel).

Important: If your kernel uses memory barriers, to minimize hardware resources, specify a value for the `reqd_work_group_size` attribute, or assign to the `max_work_group_size` attribute the smallest work-group size that accommodates all your runtime work-group size requests.

- If the kernel does not contain a barrier or refer to the local work-item ID, or if your host code does not query the work-group size, the default work-group size is the global NDRange size.

Specifying a smaller work-group size than the default at runtime might lead to excessive hardware consumption. Therefore, if you require a work-group size other than the default, specify the `max_work_group_size` attribute to set a maximum work-group size. If the work-group size remains constant through all kernel invocations, specify a required work-group size by including the `reqd_work_group_size` attribute. The `reqd_work_group_size` attribute instructs the AOC to allocate exactly the correct amount of hardware to manage the number of work-items per work-group you specify. This allocation results in hardware resource savings and more efficient kernel compute unit implementation. By specifying the `reqd_work_group_size` attribute, you also prevent the AOC from implementing additional hardware to support work-groups of unknown sizes.

For example, the code fragment below assigns a fixed work-group size of 64 work-items to a kernel:

```c
__attribute__((reqd_work_group_size(64,1,1)))
__kernel void sum (__global const float * restrict a,
                  __global const float * restrict b,
                  __global float * restrict answer)
{
    size_t gid = get_global_id(0);

    answer[gid] = a[gid] + b[gid];
}
```

Loop Unrolling

You have a lot of control in the way the Altera Offline Compiler (AOC) translates OpenCL kernel descriptions to hardware resources. If your OpenCL kernel contains loop iterations, you can increase performance by unrolling the loop. Loop unrolling decreases the number of iterations the AOC executes, at the expense of increased hardware resource consumption.
Consider the OpenCL code for a parallel application in which each work-item is responsible for computing the accumulation of four elements in an array:

```c
__kernel void example (__global const int * restrict x, __global int * restrict sum)
{
    int accum = 0;
    for (size_t i=0; i<4; i++)
    {
        accum += x[ i + get_global_id(0) * 4 ];
    }
    sum[get_global_id(0)] = accum;
}
```

Notice the three main operations that occur in this kernel:
- Load operations from input `x`
- Accumulation
- Store operations to output `sum`

The AOC arranges these operations in a pipeline according to the data flow semantics of the OpenCL kernel code. For example, the AOC implements loops by forwarding the results at the end of the pipeline back to the top of the pipeline, depending on the loop exit condition.

The OpenCL kernel performs one loop iteration of each work-item per clock cycle. With sufficient hardware resources, you can increase kernel performance by unrolling the loop, which decreases the number of iterations that the kernel executes. To unroll a loop, you add a `#pragma unroll` directive to the main loop, as shown in the kernel code sample below. Keep in mind loop unrolling significantly changes the structure of the compute unit that the AOC creates.

```c
__kernel void example (__global const int * restrict x, __global int * restrict sum)
{
    int accum = 0;

    #pragma unroll
    for (size_t i=0; i<4; i++)
    {
        accum += x[ i + get_global_id(0) * 4 ];
    }
    sum[get_global_id(0)] = accum;
}
```

In this example, the `#pragma unroll` directive causes the AOC to unroll the four iterations of the loop completely. To accomplish the unrolling, the AOC expands the pipeline by tripling the number of addition operations, and by loading four times more data. With the removal of the loop, the compute unit assumes a feed-forward structure. As a result, the compute unit can store the `sum` elements every clock cycle after the completion of the initial load operations and additions. The AOC further optimizes this kernel by coalescing the four load operations so that the compute unit can load all the necessary input data to calculate a result in one load operation.
Avoid nested looping structures. Instead, implement a large single loop or unroll inner loops by adding the \#pragma unroll directive whenever possible.

Unrolling the loop and coalescing the load operations allow the hardware implementation of the kernel to perform more operations per clock cycle. In general, the methods you use to improve the performance of your OpenCL kernels should achieve the following results:

- Increase the number of parallel operations
- Increase the memory bandwidth of the implementation
- Increase the number of operations per clock cycle that the kernels can perform in hardware

The AOC might not be able to unroll a loop completely under the following circumstances:

- You specify complete unrolling of a data-dependent loop with a very large number of iterations.
- The hardware implementation of your kernel might not fit into the FPGA.
- The loop bounds are not constants.
- The loop consists of complex control flows (for example, a loop containing complex array indexes or exit conditions that are unknown at compilation time).

For the last two cases listed above, the AOC issues the following warning:

Full unrolling of the loop is requested but the loop bounds cannot be determined. The loop is not unrolled.

To enable loop unrolling in these situations, you can limit the number of iterations the AOC unrolls by specifying the \#pragma unroll \textless N\textgreater directive, where \textless N\textgreater is the unroll factor.

You can also control the maximum amount of loop unrolling by specifying the max_unroll_loops attribute. You can override the unroll factor specified for max_unroll_loops for a particular loop by providing the \texttt{unroll} pragma. For example, to prevent a loop in your kernel from unrolling, add the directive \#pragma unroll 1 to that loop.

**Note:** You can also control the maximum amount of loop unrolling by specifying the max_unroll_loops attribute. You can override the unroll factor specified for max_unroll_loops for a particular loop by providing the \texttt{unroll} pragma. For example, to prevent a loop in your kernel from unrolling, add the directive \#pragma unroll 1 to that loop.

**Related Information**

**Single Work-Item Kernel Programming Considerations** on page 1-32
Refer to the Single Work-Item Kernel Programming Considerations section for tips on how to construct well-structured loops.

**Resource Sharing**

Share infrequently-used hardware to free up resources for implementing more single instruction multiple data (SIMD) vector lanes and compute units.

Sometimes, only a portion of the compute unit is necessary for kernel execution. The AOC can share these under-utilized sections of the compute unit to conserve hardware resources. Resource sharing might lead to hardware savings, freeing up resources for the implementation of more parallel hardware (for example, SIMD vector lanes or compute units). As the first step in optimizing resource sharing, specify the maximum amount of shared resources in your kernel by including the max_share_resources attribute. The use of max_share_resources instructs the AOC to attempt resource sharing without degrading performance. In this case, resource sharing only takes place in infrequently-used blocks. If you require additional hardware savings, consider specifying the number of shared resources using the num_share_resources attribute.
The `num_share_resources` attribute forces the AOC to share resources at the expense of overall performance.

**Kernel Vectorization**

To achieve higher throughput, you can vectorize your kernel. Kernel vectorization allows multiple work-items to execute in SIMD fashion. You can direct the AOC to translate each scalar operation in the kernel, such as addition or multiplication, to an SIMD operation.

Include the `num_simd_work_items` attribute in your kernel code to direct the AOC to perform more additions per work-item without modifying the body of the kernel. The following code fragment applies a vectorization factor of four to the original kernel code:

```c
__attribute__((num_simd_work_items(4)))
__attribute__((reqd_work_group_size(64,1,1)))
__kernel void sum (__global const float * restrict a,
                  __global const float * restrict b,
                  __global float * restrict answer)
{
    size_t gid = get_global_id(0);

    answer[gid] = a[gid] + b[gid];
}
```

**Attention:** To use the `num_simd_work_items` attribute, you must also specify a required work-group size of the kernel using the `reqd_work_group_size` attribute. The value you assign to `num_simd_work_items` must divide evenly the work-group size you specify for the `reqd_work_group_size` attribute. In the code example above, the kernel has a fixed work-group size of 64 work-items. Within each work-group, the work-items are distributed evenly among the four SIMD vector lanes. After the AOC implements the four SIMD vector lanes, each work-item now performs four times more work.

The AOC vectorizes the code and might coalesce memory accesses. You do not need to change any kernel code or host code because the AOC applies these optimizations automatically.

You can vectorize your kernel code manually, but you must adjust the NDRange in your host application to reflect the amount of vectorization you implement. The following example shows the changes in the code when you duplicate operations in the kernel manually:

```c
__kernel void sum (__global const float * restrict a,
                  __global const float * restrict b,
                  __global float * restrict answer)
{
    size_t gid = get_global_id(0);

    answer[gid * 4 + 0] = a[gid * 4 + 0] + b[gid * 4 + 0];
    answer[gid * 4 + 1] = a[gid * 4 + 1] + b[gid * 4 + 1];
    answer[gid * 4 + 2] = a[gid * 4 + 2] + b[gid * 4 + 2];
    answer[gid * 4 + 3] = a[gid * 4 + 3] + b[gid * 4 + 3];
}
```
In this form, the kernel loads four elements from arrays a and b, calculates the sums, and stores the results into the array answer. Because the FPGA pipeline loads and stores data to neighboring locations in memory, you can direct the AOC manually to coalesce each group of four load and store operations.

**Attention:** Each work-item handles four times as much work after you implement the manual optimizations. As a result, the host application must use an NDRange that is four times smaller than in the original example. On the contrary, you do not need to adjust the NDRange size when you exploit the automatic vectorization capabilities of the AOC. You can adjust the vector width with minimal code changes by using the `num_simd_work_items` attribute.

### Static Memory Coalescing

Static memory coalescing is an AOC optimization step that attempts to reduce the number of times a kernel accesses non-private memory.

The figure below shows a common case where kernel performance might benefit from static memory coalescing:

**Figure 1-5: Static Memory Coalescing**

Consider the following vectorized kernel:

```c
__attribute__((num_simd_work_items(4)))
__attribute__((reqd_work_group_size(64,1,1)))
__kernel void sum (__global const float * restrict a,
__global const float * restrict b,
__global float * restrict answer)
{
    size_t gid = get_global_id(0);
    answer[gid] = a[gid] + b[gid];
}
```
answer[gid] = a[gid] + b[gid];
}

The OpenCL kernel performs four load operations that access consecutive locations in memory. Instead of performing four memory accesses to competing locations, the AOC coalesces the four loads into a single, wider vector load. This optimization reduces the number of accesses to a memory system and potentially leads to better memory access patterns.

Although the AOC performs static memory coalescing automatically when it vectorizes the kernel, you should use wide vector loads and stores in your OpenCL code whenever possible to ensure efficient memory accesses. To implement static memory coalescing manually, you must write your code in such a way that a sequential access pattern can be identified at compilation time. The original kernel code shown in the figure above can benefit from static memory coalescing because all the indexes into buffers a and b increment with offsets that are known at compilation time. In contrast, the following code does not allow static memory coalescing to occur:

__kernel void test (__global float * restrict a,
                          __global float * restrict b,
                          __global float * restrict answer;
                          __global int * restrict offsets)
{
        size_t gid = get_global_id(0);

        answer[gid*4 + 0] = a[gid*4 + 0 + offsets[gid]] + b[gid*4 + 0];
        answer[gid*4 + 1] = a[gid*4 + 1 + offsets[gid]] + b[gid*4 + 1];
        answer[gid*4 + 2] = a[gid*4 + 2 + offsets[gid]] + b[gid*4 + 2];
        answer[gid*4 + 3] = a[gid*4 + 3 + offsets[gid]] + b[gid*4 + 3];
}

In this example, the value offsets[gid] is unknown at compilation time. As a result, the AOC cannot statically coalesce the read accesses to buffer a.

Multiple Compute Units

To achieve higher throughput, the AOC can generate multiple compute units for each kernel. The AOC implements each compute unit as a unique pipeline. Generally, each kernel compute unit can execute multiple work-groups simultaneously.

To increase overall kernel throughput, the hardware scheduler in the FPGA dispatches work-groups to additional available compute units. A compute unit is available for work-group assignments as long as it has not reached its full capacity.

Assume each work-group takes the same amount of time to complete its execution. If the AOC implements two compute units, each compute unit executes half of the work-groups. Because the hardware scheduler dispatches the work-groups, you do not need to manage this process in your own code.

The AOC does not determine automatically the optimal number of compute units for a kernel. To increase the number of compute units for your kernel implementation, you must specify the number of compute units the AOC should create with the num_compute_units attribute, as shown in the code sample below.

__attribute__((num_compute_units(2)))
__kernel void sum (__global const float * restrict a,
__global const float * restrict b, __global float * restrict answer)
{
    size_t gid = get_global_id(0);
    answer[gid] = a[gid] + b[gid];
}

As shown in the figure below, increasing the number of compute units achieves higher throughput. However, you do so at the expense of increasing global memory bandwidth among the compute units, and increasing hardware resource utilization.

Figure 1-6: Data Flow with Multiple Compute Units

Compute Unit Replication versus Kernel SIMD Vectorization

In most cases, you should implement the num_simd_work_items attribute to increase data processing efficiency before using the num_compute_units attribute.

Both the num_compute_units and num_simd_work_items attributes increase throughput by increasing the amount of hardware the AOC uses to implement your kernel. The num_compute_units attribute modifies the number of compute units to which work-groups can be scheduled. In contrast, the num_simd_work_items attribute modifies the amount of work a compute unit can perform in parallel on a single work-group. The num_simd_work_items attribute duplicates only the datapath of the compute unit by sharing the control logic across each SIMD vector lane.

Generally, using the num_simd_work_items attribute leads to more efficient hardware than using the num_compute_units attribute to achieve the same goal. The num_simd_work_items attribute also allows the AOC to coalesce your memory accesses.

Oftentimes, when using the num_compute_units attribute to replicate the kernel compute unit, you also increase the number of times the kernel accesses global memory. In contrast, implementing the num_simd_work_items attribute might simply widen the datapaths of the load and store operations, as shown in the figure below.
Multiple compute units competing for global memory might lead to undesired memory access patterns. You can alter the undesired memory access pattern by introducing the `num_simd_work_items` attribute instead of the `num_compute_units` attribute. In addition, the `num_simd_work_items` attribute potentially offers the same computational throughput as the equivalent kernel compute unit duplication that the `num_compute_units` attribute offers.

You cannot implement the `num_simd_work_items` attribute in your kernel under the following circumstances:

- The value you specify for `num_simd_work_items` is not 2, 4, 8 or 16.
- The value of `reqd_work_group_size` is not divisible by `num_simd_work_items`.

  For example, the following declaration is incorrect because 50 is not divisible by 4:

  ```
  __attribute__((num_simd_work_items(4)))
  __attribute__((reqd_work_group_size(50,0,0)))
  ```

- Kernels with complex control flows. You cannot vectorize kernels in which different work-items follow different control paths (for example, the control paths depend on `get_global_ID` or `get_local_ID`).

  During kernel compilation, the AOC issues messages informing you whether the implementation of vectorization optimizations is successful. Kernel vectorization is successful if the reported vectorization factor matches the value you specify for the `num_simd_work_items` attribute.

### Combination of Compute Unit Replication and Kernel SIMD Vectorization

Sometimes, your replicated or vectorized kernel might not fit in the FPGA. In that case, you can modify the kernel by both replicating the compute unit and vectorizing the kernel. You include `num_compute_units` to modify the number of compute units for the kernel, and include `num_simd_work_items` to take advantage of kernel vectorization.
Consider a case where a kernel with a `num_simd_work_items` attribute set to 16 does not fit in the FPGA. The kernel might fit if you modify it by duplicating a narrower SIMD kernel compute unit. Determining the optimal balance between the number of compute units and the SIMD width might require some experimentation. For example, duplicating a four lane-wide SIMD kernel compute unit three times might achieve better throughput than duplicating an eight lane-wide SIMD kernel compute unit twice.

The following example code shows how you can combine the `num_compute_units` and `num_simd_work_items` attributes in your OpenCL code:

```c
__attribute__((num_simd_work_items(4)))
__attribute__((num_compute_units(3)))
__attribute__((reqd_work_group_size(8,8,1)))
__kernel void matrixMult(__global float * restrict C,
                         __global float * restrict A,
                         ...
```

The figure below illustrates the data flow of the kernel described above. The `num_compute_units` implements three replicated compute units. The `num_simd_work_items` implements four SIMD vector lanes.

**Figure 1-8: Optimizing Throughput by Combining Compute Unit Replication and Kernel SIMD Vectorization**

**Attention:** You can also enable the resource-driven optimizer to determine automatically the best combination of `num_compute_units` and `num_simd_work_items`.

**Important:** It is more time-consuming to compile a hardware design that fills the entire FPGA than smaller designs. When you adjust your kernel optimizations, remove the increased number of SIMD vector lanes and compute units prior to recompiling the kernel.

**Resource-Driven Optimization**

To optimize throughput, you can enable the resource-driven optimizer to direct the AOC to analyze automatically the effects of combining various values of kernel attributes.
Capability of the Resource-Driven Optimizer

The resource-driven optimizer is a feature of the AOC. It examines multiple values of unspecified kernel attributes in various combinations, and applies a set of heuristics to improve a base design incrementally. During compilation, the resource-driven optimizer identifies a set of values for the following kernel attributes:

- num_compute_units
- num_simd_work_items
- max_unroll_loops
- max_share_resources
- num_share_resources

The AOC implements this set of values to maximize kernel performance in terms of work-items executed per second.

Based on the result of its analysis, the AOC optimizes code blocks that work-items execute frequently. For these code blocks, the AOC uses additional hardware resources to achieve an implementation with higher throughput. For code blocks that work-items execute infrequently, the AOC attempts to reuse the same hardware to implement multiple operations.

The amount of hardware sharing that occurs is called the sharing degree. It is the number of times an operation is shared by work-items executing within the same compute unit. Code blocks that work-items execute infrequently might lead to a higher sharing degree.

The AOC does not modify values of kernel attributes or pragmas that you specify in kernel declarations. The AOC modifies only unspecified attributes and pragmas.

Optimization Behavior

The resource-driven optimizer implements the following optimizations:

- The optimizer attempts resource sharing of infrequently-executed code blocks only if the kernel does not fit the FPGA.
  
  After the AOC identifies an optimized kernel that fits within the FPGA, the optimizer applies optimizations that increase performance.

- If a design contains multiple kernels, the optimizer improves the kernel (or kernels) with minimum performance first.
  
  The optimizer determines which kernels to optimize first based on the work-items per second metric. When the optimizer cannot optimize these kernels any further, it improves subsequent kernels, in order of their throughput estimates. For each kernel, the optimizer maintains a set of high-performance candidates and attempts to apply incremental optimizations to each of them. The optimizer favors loop unrolling and SIMD vectorization over compute unit replication because these optimizations generally result in more efficient hardware implementations.

- The optimizer iterates on a predetermined set of optimization steps.
  
  In many cases, the optimizer infers optimization ranges ahead of time. For example, the optimizer determines the maximum number of compute units based on the available memory bandwidth. Anytime the optimizer fails to perform an optimization, it skips that step and attempts other optimizations.

Usage

To enable the resource-driven optimizer, at a command prompt, type the command

```
aoc -O3 --util <maximum_logic_utilization> <your_kernel_filename>.cl
```
where `<maximum_logic_utilization>` is the maximum hardware resource on the FPGA that the kernel uses.

If you do not specify the maximum logic utilization, the resource-driven optimizer uses a maximum logic utilization of 85%. To enable the resource-driven optimizer with a maximum FPGA hardware utilization of 85%, omit the `--util` flag, as shown below:

```
aoc -O3 <your_kernel_filename>.cl
```

To specify a maximum FPGA hardware utilization percentage other than 85% (for example, 50%), invoke the following command:

```
aoc -O3 --util 50 <your_kernel_filename>.cl
```

Each combination of kernel attributes that the optimizer adjusts is called a design point. The optimizer prints a summary of each design point it explores. For each design point, the optimizer lists the area utilization, and the attribute settings for each kernel.

**Caution:** If you specify a maximum logic utilization value that exceeds 85%, compilation might fail at the hardware compilation stage. Alternatively, compilation might take a long time to complete because the requested amount of FPGA hardware resources is too great.

**Limitations**

Static optimizations are subjected to some inherent limitations. The control flow analyses assume values of kernel arguments, passed from the host, that are unknown at compilation time. For example, the optimizer assumes that loops with unknown bounds iterate 1024 times. Based on these assumptions, the optimizer might guide the optimizations towards code blocks that work-items execute less often than estimated. In the case of loops with unknown bounds, you can override the amount of unrolling by specifying an unroll factor in the code using the `unroll` pragma. If you want to prevent the resource-driven optimizer from unrolling a loop, you can specify an unroll factor of 1 to indicate no loop unrolling.

Another limiting factor is that all optimizations take place before hardware compilation occurs. The performance estimation might not capture accurately the maximum operating frequency that the hardware compiler achieves. Also, the optimizer relies on an estimation of the actual resource usage. If the compiled hardware design uses a different amount of resources, you can change the target utilization threshold to compensate for any discrepancy. You do so by specifying a different logic utilization amount via the `--util <maximum_logic_utilization>` option.

There are also range limitations on the amount of sharing and vectorization. Currently, the maximum sharing degree is 8, and the maximum number of SIMD vector lanes is 16.

**Floating-Point Operations**

For floating-point operations, you have the option to direct the AOC to perform optimizations that create more efficient pipeline structures in hardware and reduce the overall hardware usage. These optimizations can cause small differences in floating-point results. Therefore, you must enable these optimization options manually.

**Tree Balancing**

Order of operation rules apply in the OpenCL language. In the following example, the AOC performs multiplications and additions in a strict order, performing operations within the innermost parentheses first:

```
result = (((A * B) + C) + (D * E)) + (F * G);
```
By default, the AOC creates an implementation that resembles a long vine for such computations, as shown in the figure below:

**Figure 1-9: Default Floating-Point Implementation**

Long, unbalanced operations lead to more expensive hardware. A more efficient hardware implementation is a *balanced tree*, shown in figure below:

**Figure 1-10: Balanced Tree Floating-Point Implementation**

In a balanced tree implementation, the AOC converts the long vine of floating-point adders into a tree pipeline structure. The AOC does not perform tree balancing of floating-point operations automatically because the outcomes of the floating-point operations might differ. As a result, this optimization is inconsistent with the IEEE Standard 754-2008.

If you want the AOC to optimize floating-point operations by using balanced trees, and your program can tolerate small differences in floating-point results, include the `-fp-relaxed` option in the `aoc` command, as shown below:

```
aoc -fp-relaxed=true <your_kernel_filename>.cl
```
Rounding Operations

The balanced tree implementation of a floating-point operation includes multiple rounding operations. These rounding operations can require a significant amount of hardware resources in some applications. The AOC does not reduce the number of rounding operations automatically because doing so violates the results required by IEEE Standard 754-2008.

You can reduce the amount of hardware necessary to implement floating-point operations with the -fpc argument of the aoc command. The -fpc argument directs the AOC to perform the following tasks:

- Remove floating-point rounding operations and conversions whenever possible.
- If possible, the -fpc argument directs the AOC to round a floating-point operation only once—at the end of the tree of the floating-point operations.
- Carry additional mantissa bits to maintain precision.

The AOC carries additional precision bits through the floating-point calculations, and removes these precision bits at the end of the tree of floating-point operations.

This type of optimization results in hardware that is often called a fused floating-point operation, and it is a feature of many new hardware processing systems. Fusing multiple floating-point operations minimizes the number of rounding steps, which leads to more accurate results. An example of this optimization is a fused multiply and accumulate (FMAC) instruction available in new processor architectures. Unlike most other architectures, the AOC can provide fused floating-point mathematical capabilities for many combinations of floating-point operators in your kernel.

If you want the AOC to optimize floating-point operations by reducing the number of rounding operations, and your program can tolerate small differences in floating-point results, invoke the following command:

aoc -fpc=true <your_kernel_filename>.cl

Floating-Point versus Fixed-Point Representations

An FPGA contains a substantial amount of logic for implementing floating-point operations. However, you can increase the amount of hardware resources available by using a fixed-point representation of the data whenever possible. The hardware necessary to implement a fixed-point operation is typically smaller than the equivalent floating-point operation. As a result, you can fit more fixed-point operations into an FPGA than the floating-point equivalent.

The OpenCL standard does not support fixed-point representation; you must implement fixed-point representations using integer data types. Hardware developers commonly achieve hardware savings by using fixed-point data representations and only retain a data resolution required for performing calculations. You must use an 8, 16, 32, or 64-bit scalar data type because the OpenCL standard supports only these data resolutions. However, you can incorporate the appropriate masking operations in your source code so that the hardware compilation tools can perform optimizations to conserve hardware resources.

For example, if an algorithm uses a fixed-point representation of 17-bit data, you must use a 32-bit data type to store the value. If you then direct the AOC to add two 17-bit fixed-point values together, the AOC must create extra hardware to handle the addition of the excess upper 15 bits. To avoid having this additional hardware, you can use static bit masks to direct the hardware compilation tools to disregard the unnecessary bits during hardware compilation. The code below implements this masking operation:

```c
__kernel fixed_point_add(__global const unsigned int * restrict a,
                   __global const unsigned int * restrict b,
                   __global unsigned int * restrict result)
{
```
size_t gid = get_global_id(0);

unsigned int temp;
temp = 0x3_FFFF & ((0x1_FFFF & a[gid]) + ((0x1_FFFF & b[gid])));
result[gid] = temp & 0x3_FFFF;
}

In this code example, the upper 15 bits of inputs a and b are masked away and added together. Because the result of adding two 17-bit values cannot exceed an 18-bit resolution, the AOC applies an additional mask to mask away the upper 14 bits of the result. The final hardware implementation is a 17-bit addition as opposed to a full 32-bit addition. The logic savings in this example are relatively minor compared to the sheer number of hardware resources available in the FPGA. However, these small savings, if applied often, can accumulate into a larger hardware saving across the entire FPGA device.

Optimization of Memory Access Efficiency

Memory access efficiency often dictates the overall performance of your OpenCL kernel. When developing your OpenCL code, it is advantageous to minimize the number of global memory accesses. The OpenCL Specification version 1.0 describes four memory types: global, constant, local, and private memories.

An interconnect topology connects shared global, constant, and local memory systems to their underlying memory.

Memory accesses compete for shared memory resources (that is, global, local, and constant memories). If your OpenCL kernel performs a large number of memory accesses, the AOC must generate complex arbitration logic to handle the memory access requests. The complex arbitration logic might cause a drop in the maximum operating frequency ($F_{\text{max}}$), which degrades kernel performance.

The following sections discuss memory access optimizations in detail. In summary, minimizing global memory accesses is beneficial for the following reasons:

- Typically, increases in OpenCL kernel performance lead to increases in global memory bandwidth requirements.
- The maximum global memory bandwidth is much smaller than the maximum local memory bandwidth.
- The maximum computational bandwidth of the FPGA is much larger than the global memory bandwidth.

**Attention:** Use local, private or constant memory whenever possible to increase the memory bandwidth of the kernel.

1. **General Guidelines on Optimizing Memory Accesses** on page 1-24
   Optimizing the memory accesses in your OpenCL kernels can improve overall kernel performance.

2. **Optimize Global Memory Accesses** on page 1-24
   The AOC interleaves global memory across each of the external memory banks.

3. **Perform Kernel Computations Using Constant, Local or Private Memory** on page 1-28
   To optimize memory access efficiency, minimize the number for global memory accesses by performing your OpenCL kernel computations in constant, local, or private memory.
4. **Single Work-Item Execution** on page 1-30

You can direct the AOCL host to execute kernels in a single work-item, which is equivalent to launching a kernel with an NDRRange of \((1, 1, 1)\).

**General Guidelines on Optimizing Memory Accesses**

Optimizing the memory accesses in your OpenCL kernels can improve overall kernel performance.

Consider implementing the following techniques for optimizing memory accesses, whenever possible:

- If your OpenCL program has a pair of kernels—one produces data and the other one consumes that data—convert them into a single kernel that performs both functions. Also, implement helper functions to separate logically the functions of the two original kernels.
  
  FPGA implementations favor one large kernel over separate smaller kernels. Kernel unification removes the need to write the results from one kernel into global memory temporarily before fetching the same data in the other kernel.

- The AOC implements local memory in FPGAs very differently than in GPUs. If your OpenCL kernel contains code to avoid GPU-specific local memory bank conflicts, remove that code because the AOC generates hardware that avoids local memory bank conflicts automatically whenever possible.

**Optimize Global Memory Accesses**

The AOC uses SDRAM as global memory. By default, the AOC configures global memory in a burst-interleaved configuration. The AOC interleaves global memory across each of the external memory banks.

In most circumstances, the default burst-interleaved configuration leads to the best load balancing between the memory banks. However, in some cases, you might want to partition the banks manually as two non-interleaved (and contiguous) memory regions to achieve better load balancing.

The figure below illustrates the differences in memory mapping patterns between burst-interleaved and non-interleaved memory partitions.
Contiguous Memory Accesses

Contiguous memory access optimizations analyze statically the access patterns of global load and store operations in a kernel. For sequential load or store operations that occur for the entire kernel invocation, the AOC directs the kernel to access consecutive locations in global memory.

Consider the following code example:

```c
__kernel void sum ( __global const float * restrict a,
                   __global const float * restrict b,
                   __global float * restrict c )
{
  size_t gid = get_global_id(0);
  c[gid] = a[gid] + b[gid];
}
```

The load operation from array `a` uses an index that is a direct function of the work-item global ID. By basing the array index on the work-item global ID, the AOC can direct contiguous load operations. These load operations retrieve the data sequentially from the input array, and sends the read data to the pipeline as required. Contiguous store operations then store elements of the result exiting the computation pipeline in sequential locations within global memory.

**Tip:** Use the `const` qualifier for any read-only global buffer so that the AOC can perform more aggressive optimizations on the load operation.
The following figure illustrates an example of the contiguous memory access optimization:

**Figure 1-12: Contiguous Memory Access**

Contiguous load and store operations improve memory access efficiency because they lead to increased access speeds and reduced hardware resource needs. The data travels in and out of the computational portion of the pipeline concurrently, allowing overlaps between computation and memory accesses. If possible, use work-item IDs that index consecutive memory locations for load and store operations that access global memory. Sequential accesses to global memory increase memory efficiency because they provide an ideal access pattern.

**Manual Partitioning of Global Memory**

You can partition the memory manually so that each buffer occupies a different memory bank.

The default burst-interleaved configuration of the global memory prevents load imbalance by ensuring that memory accesses do not favor one external memory bank over another. However, you have the option to control the memory bandwidth across a group of buffers by partitioning your data manually.

- To partition global memory manually, compile your OpenCL kernels with the `--sw-dimm-partition` flag to configure the memory banks as separate (non-interleaved) banks.

  If your kernel accesses two buffers of equal size in memory, you can distribute your data to both memory banks simultaneously regardless of dynamic scheduling between the loads. This optimization step might increase your apparent memory bandwidth.

- The AOC cannot burst-interleave across different memory types. To partition heterogeneous global memory types manually, compile your OpenCL kernels with the `--no-interleaving <memory_type>` flag to configure each bank of a certain memory type as non-interleaved banks.

**Related Information**

*Altera SDK for OpenCL Programming Guide*

For more information on the usage of the `--sw-dimm-partition` flag, refer to the `--sw-dimm-partition` section of the AOCL Programming Guide.
Heterogeneous Memory Buffers

You can execute your kernel on an FPGA board that includes multiple global memory types, such as DDR, quad data rate (QDR), and on-chip RAMs.

**Important:** The AOCL version 13.1 offers heterogenous memory support as a beta feature.

If your FPGA board offers heterogeneous global memory types, keep in mind that they handle different memory accesses with varying efficiencies.

For example:

- Use DDR SDRAM for long sequential accesses.
- Use QDR SDRAM for random accesses.
- Use on-chip RAM for random low latency accesses.

Your kernel can access heterogeneous memory types by allocating buffers in the global memory space. Include the `buffer_location` attribute in your kernel code to define the memory type in which the buffer is allocated.

Altera recommends that you define the `buffer_location` attribute in a preprocessor macro for ease of reuse, as shown below:

```c
#define QDR__global__attribute__((buffer_location("QDR")))
#define DDR__global__attribute__((buffer_location("DDR")))
__kernel void foo (QDR uint * data, DDR uint * lup)
{
   //statements
}
```

**Attention:** If you assign a kernel argument to a non-default memory (for example, QDR uint * data and DDR uint * lup from the code above), you cannot declare that argument using the `const` keyword. In addition, you cannot perform atomic operations with pointers derived from that argument.

Host Application Modifications for Heterogeneous Memory Accesses

By default, the host allocates buffers into the main memory when you load kernels into the OpenCL runtime via the `clCreateProgramWithBinary` function. As a result, the host relocates heterogenous memory buffers that are bound to kernel arguments to the main memory automatically upon kernel invocation. To avoid the initial allocation of heterogeneous memory buffers in the main memory, include the `CL_MEM_HETEROGENEOUS_ALTERA` flag when you use the `clCreateBuffer` function to allocate memory, as shown below:

```c
mem = clCreateBuffer(context,
   flags | CL_MEM_HETEROGENEOUS_ALTERA,
   memSize,
   ...)
```
If a memory (`cl_mem`) object residing in a memory type is set as a kernel argument that corresponds to a different memory technology, the host moves the memory object automatically when it queues the kernel. **Caution:** Do not pass a buffer as kernel arguments that associate it with multiple memory technologies.

**Related Information**

[Altera SDK for OpenCL Programming Guide](#)

For more information on the tools available for heterogeneous memory support, refer to the `--no-interleaving` and Kernels Pragmas and Attributes sections of the AOCL Programming Guide.

**Perform Kernel Computations Using Constant, Local or Private Memory**

To optimize memory access efficiency, minimize the number for global memory accesses by performing your OpenCL kernel computations in constant, local, or private memory.

To minimize global memory accesses, you must first preload data from a group of computations from global memory to constant, local, or private memory. You perform the kernel computations on the preloaded data, and then write the results back to global memory.

**Constant Cache Memory**

Constant memory resides in global memory, but the kernel loads it into an on-chip cache shared by all work-groups at runtime. For example, if you have read-only data that all work-groups use, and the data size of the constant buffer fits into the constant cache, allocate the data to the constant memory. The constant cache is most appropriate for high-bandwidth table lookups that are constant across several invocations of a kernel. The constant cache is optimized for high cache hit performance.

By default, the constant cache size is 16 kB. You can specify the constant cache size by including the `--const-cache-bytes <N>` flag in your `aoc` command, where `<N>` is the constant cache size in bytes.

Unlike global memory accesses that have extra hardware for tolerating long memory latencies, the constant cache suffers large performance penalties for cache misses. If the `__constant` arguments in your OpenCL kernel code cannot fit in the cache, you might achieve better performance with `__global const` arguments instead. If the host application writes to constant memory already loaded into the constant cache, the cached data is discarded (that is, invalidated) from the constant cache.

**Preloading Data to Local Memory**

Local memory is considerably smaller than global memory; however, it has significantly higher throughput and much lower latency. Compared to global memory, the kernel can access local memory randomly without any performance penalty. In your kernel code, attempt to access the global memory sequentially, and buffer that data in on-chip local memory before your kernel uses the data for calculation purposes.

The AOC implements OpenCL local memory in on-chip memory blocks in the FPGA. On-chip memory blocks have two read and write ports, and can be clocked at an operating frequency that is double the operating frequency of the OpenCL kernels. This doubling of the clock frequency allows the memory to be “double pumped,” resulting in twice the bandwidth from the same memory. As a result, each on-chip memory block supports up to four simultaneous accesses.
Ideally, the accesses to each bank are distributed uniformly across the on-chip memory blocks of the bank. Because only four simultaneous accesses to an on-chip memory block are possible in a single clock cycle, distributing the accesses helps avoid bank contention.

This banking configuration is usually effective; however, the AOC must create a complex memory system to accommodate a large number of banks. A large number of banks might complicate the arbitration network and can reduce the overall system performance.

Because the AOC implements local memory that resides in on-chip memory blocks in the FPGA, the AOC must choose the size of local memory systems at compilation time. The method the AOC uses to determine the size of a local memory system depends on the local data types used in your OpenCL code.

Optimizing Local Memory Accesses

To optimize local memory access efficiency, consider the following guidelines:

- Implementing certain optimizations techniques, such as loop unrolling, might lead to more concurrent memory accesses.
  
  **Caution:** Increasing the number of memory accesses can complicate the memory systems and degrade performance.

- Simplify the local memory subsystem by limiting the number of unique local memory accesses in your kernel to four or less, whenever possible.

  You achieve maximum local memory performance when there are four or less memory accesses to a local memory system. If the number of accesses to a particular memory system is greater than four, the AOC arranges the on-chip memory blocks of the memory system into a banked configuration.

- If you have function scope local data, the AOC statically sizes the local data that you define within a function body at compilation time. You should define local memories by directing the AOC to set the memory to the required size, rounded up to a power of two.

  For pointers to __local kernel arguments, the host assigns their memory sizes dynamically at runtime through clSetKernelArg calls. However, the AOC must set these physical memory sizes at compilation time.

  By default, pointers to __local kernel arguments are 16 kB in size. You can specify an allocation size by including the local_mem_size attribute in your pointer declaration.

  **Note:** clSetKernelArg calls can request a smaller data size than has been physically allocated at compilation time, but never a larger size.

Related Information

**Altera SDK for OpenCL Programming Guide**

For more information on the usage of the local_mem_size attribute, refer to the Kernel Pragmas and Attributes section of the AOCL Programming Guide.

Storing Variables and Arrays in Private Memory

The AOC implements private memory using FPGA registers. Typically, private memory is useful for storing single variables or small arrays. Registers are plentiful hardware resources in FPGAs, and it is almost always better to use private memory instead of other memory types whenever possible. The kernel can access private memories in parallel, allowing them to provide more bandwidth than any other memory type (that is, global, local, and constant memories).
Single Work-Item Execution

You can direct the AOCL host to execute kernels in a single work-item, which is equivalent to launching a kernel with an NDRange of (1, 1, 1). The OpenCL Specification version 1.0 describes this mode of operation as task parallel programming. A task refers to a kernel executed on a compute unit consisting of one work-group that contains one work-item.

Generally, the host executes multiple work-items in parallel to compute OpenCL kernel instructions. However, the data parallel programming model is not suitable for situations where the compiler must partition fine-grained data among parallel work-items. In these cases, you can maximize throughput by expressing your kernel as a single work-item. Unlike NDRange kernels, single work-item kernels follow a more natural sequential model similar to C programming. Particularly, you do not have to partition the data across work-items.

To ensure high-throughput task-based kernel execution on the FPGA, the AOC must process multiple pipeline stages in parallel at any given time. The mode of operation is particularly challenging in loops because by default, the AOC executes loop iterations sequentially through the pipeline.

Consider the following code example:

```c
float ai[32];
for (int i = 0; i < stream_size; i++)
{
    float fir = 0;
    ai[0] = input[i];

    #pragma unroll 31
    for (int k = 31; k > 0; k--)
    {
        fir += ai[k] * coeff[k];
        ai[k] = ai[k-1];
    }
    fir += ai[0] * coeff[0];
    if (i >= 31)
    {
        output[i-31] = fir;
    }
}
```

During each loop iteration, data values shift into the array `ai`, and then a reduction occurs for each element of the array. In data parallel programming, an NDRange kernel requires an intricate mechanism involving local buffers and barrier constructs to pass the shifted `ai` values to each work-item, which leads to suboptimal kernel performance. However, if you rewrite the kernel as an OpenCL task, the AOC can extract the parallelism between each loop iteration. The AOC shares the shifted `ai` as a loop-carried dependency. The AOC transfers the array values between loop iterations through registers on the FPGA, allowing it to execute each loop iteration in a true pipelined fashion.

Pipeline parallel execution saturates the kernel pipeline with multiple loop iterations, allowing the AOC to process the loop in a high-throughput fashion. This loop execution model is known as loop pipelining. You must optimize your kernel for loop pipelining manually to extract the parallelism between loop iterations.

To enable loop pipelining, you must include the `task` attribute in your kernel.
Limitations

Keep in mind of the following limitations when you implement the task attribute in your kernel:

- The OpenCL task parallel programming model does not support the notion of a barrier because a task implements single-work-item execution. You must replace barriers (barrier) with memory fences (mem_fence) in your kernel. If you include barriers in a pipelined loop of an OpenCL task, the compiler errors out.
- The AOCL might generate incorrect kernel throughput estimates for OpenCL tasks.

Related Information

Altera SDK for OpenCL Programming Guide
For more information on the usage of the task kernel attribute, refer to the Kernels Pragmas and Attributes section of the AOCL Programming Guide.

Performance Warning Messages

When compiling single work-item OpenCL kernels, the AOC issues warning messages for the loops it extracts for parallel execution.

For all the loops the AOC extracts for parallel execution, the AOC issues the following warning message:

Compiler Warning: Inferring parallel execution for iterations in loop <loop_label>

where <loop_label> is a low-level label assignment for the start of a loop. It does not correspond to the labels in your OpenCL kernel source code.

Sometimes, a loop extracted for pipeline parallel execution contains memory or data dependencies, which might degrade the performance of the single work-item kernel. Consider the following code example:

```c
for (i = 0; i < N; i++)
{
    //statements
    for (j = 0; j < M; j++)
    {
        sum += N;
        //statements
    }
    //statements
}
```

If sum is an integer, and the arithmetic operation only involves integers, the AOC can extract pipeline parallelism from this loop effectively. However, if the operation involves floating-point values, pipeline stalls might occur between iterations. Because a floating-point operation is more complex than an integer operation, it often requires several FPGA clock cycles to complete. As a result, subsequent loop iterations cannot proceed until the operation from the current iteration completes.

For pipeline parallel execution involving floating-point operations, the AOC issues the following warning message to notify you that the loop labeled <loop_label> cannot execute in a pipeline parallel fashion at full efficiency:

Compiler Warning: Parallel execution at <N>% of pipeline throughput in basic block <loop_label> in function <function_name>

or
Compiler Warning: Parallel execution, successive iterations launched every \(<cycle\_count>\) cycles in basic block \(<loop\_label>\) in function \(<function\_name>\)

where \(<cycle\_count>\) is a measure of the pipeline stall in terms of FPGA clock cycles in between executions of successive iterations of the loop.

Your single work-item kernel might contain loop-carried memory dependencies that span multiple pipelined loops. If this is the case, the AOC has to generate hardware that crosses these loops to account for these memory dependencies, and it issues the following warning message:

Compiler Warning: Performance degradation due to \(<N>\) loop-carried dependencies spanning multiple basic blocks in loop \(<loop\_label>\) in function \(<function\_name>\)

where \(<N>\) is the number of loop-carried memory dependencies.

Consider the following code example:

```c
for (i = 0; i < N; i++)
{
    //statements
    for (j = 0; j < M; j++)
    {
        //statements
    }
    //statements
}
```

In this example, a loop-carried dependence exists on the outer loop, creating a loop-carried memory dependency that spans both pipelined loops.

**Single Work-Item Kernel Programming Considerations**

If your OpenCL kernels contain loop structures, construct them in a way that allows the AOC to analyze them effectively. Well-structured loops are particularly important when you direct the AOC to perform pipeline parallelism execution in loops using the `task` kernel attribute.

Follow the guidelines below to optimize loop structures in your kernel:

**Avoid Pointer Aliasing**

Insert the `restrict` keyword in pointer arguments whenever possible. Including the `restrict` keyword in pointer arguments prevents the AOC from creating unnecessary memory dependencies between non-conflicting read and write operations. Consider a loop where each iteration reads data from one array, and then it writes data to another array in the same physical memory. Without including the `restrict` keyword in these pointer arguments, the AOC might assume dependence between the two arrays, and extracts less pipeline parallelism as a result.

**Construct "Well-Formed" Loops**

A "well-formed" loop has an exit condition that compares against an integer bound, and has a simple induction increment of one per iteration. Including "well-formed" loops in your kernel improves performance because the AOC can analyze these loops efficiently.
The following example is a "well-formed" loop:

```c
for(i=0; i < N; i++)
{
    //statements
}
```

**Important:** "Well-formed" nested loops also contribute to maximizing kernel performance.

The following example is a "well-formed" nested loop structure:

```c
for(i=0; i < N; i++)
{
    //statements
    for(j=0; j < M; j++)
    {
        //statements
    }
}
```

**Minimize Loop-Carried Dependencies**

The loop structure below creates a loop-carried dependence because each loop iteration reads data written by the previous iteration. As a result, each read operation cannot proceed until the write operation from the previous iteration completes. The presence of loop-carried dependencies decreases the extent of pipeline parallelism the AOC can achieve, which decreases the performance of the OpenCL application.

```c
for(int i = 0; i < N; i++)
{
}
```

The AOC performs a static memory dependence analysis on loops to determine the extent of parallelism it can achieve. In some cases, the AOC might assume dependence between two array accesses, and extracts less pipeline parallelism as a result. The AOC assumes loop-carried dependence if it cannot resolve the dependencies at compilation time because of unknown variables, or if the array accesses involve complex addressing.
To minimize loop-carried dependencies, following the guidelines below whenever possible:

- **Avoid pointer arithmetic.**
  
  Accessing arrays by dereferencing pointer values derived from arithmetic operations results in suboptimal compiler output. For example, avoid accessing an array in the following manner:

  ```c
  for(int i = 0; i < N; i++)
  {
    int t = *(A++);
    *A = t;
  }
  ```

- **Introduce simple array indexes.**
  
  Avoid the following types of complex array indexes because the AOC cannot analyze them effectively, which might lead to suboptimal compiler output:
  - Nonconstants in array indexes.
    For example, `A[K + i]`, where `i` is the loop index variable and `K` is an unknown variable.
  - Multiple index variables in the same subscript location.
    For example, `A[i + 2 * j]`, where `i` and `j` are loop index variables for a double nested loop.
    **Note:** The AOC can analyze the array index `A[i][j]` effectively because the index variables are in different subscripts.
  - Nonlinear indexing.
    For example, `A[i & C]`, where `i` is a loop index variable and `C` is a constant or a nonconstant variable.
  
  - **Use loops with constant bounds in your kernel whenever possible.**

    Loops with constant bounds allow the AOC to perform range analysis effectively.

**Avoid Complex Loop Exit Conditions**

The AOC evaluates exit conditions to determine if subsequent loop iterations can enter the loop pipeline. There are times when the AOC requires memory accesses or complex operations to evaluate the exit condition. In these cases, subsequent iterations cannot launch until the evaluation completes, decreasing overall loop performance.

**Convert Nested Loops into a Single Loop**

To maximize performance, combine nested loops into a single form whenever possible. Restructuring nested loops into a single loop reduces hardware footprint and computational overhead between loop iterations.
The following code examples illustrate the conversion of a nested loop into a single loop:

<table>
<thead>
<tr>
<th>Nested Loop</th>
<th>Converted Single Loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (i = 0; i &lt; N*M; i++)</td>
<td>for (i = 0; i &lt; N*M; i++)</td>
</tr>
<tr>
<td>{</td>
<td>{</td>
</tr>
<tr>
<td>//statements</td>
<td>//statements</td>
</tr>
<tr>
<td>for (j = 0; j &lt; M; j++)</td>
<td>//statements</td>
</tr>
<tr>
<td>{</td>
<td>}</td>
</tr>
<tr>
<td>//statements</td>
<td>//statements</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

**Initialize Data Prior to Usage in a Loop**

To minimize the amount of loop-carried dependencies, initialize variables and arrays prior to using them in a loop. The AOC might lose the scope of variables as a result of pipeline generation.

Consider the following nested loop example:

```c
for (i = 0; i < N; i++) {
    //statements
    for (j = 0; j < M; j++) {
        int data = 0;
        if (cond) {
            data += N;
        }
        //statements
    }
    //statements
}
```

Despite the initialization of the variable `data` occurs in the inner loop, the AOC might carry the variable across the outer loop because the AOC does not preserve the scoping information for the variable `data`.

A more efficient way of initializing data used in a loop is shown below:

```c
for (i = 0; i < N; i++) {
    #pragma unroll
    for (ishift = S-1; ishift >= 0; ishift--)
        shift_reg[ishift] = 0;
    for (j = 0; j < M; j++) {
        #pragma unroll
        for (ishift = S-1; ishift > 0; ishift--)
            ...
```
In this example, the assignment of the shift register `shift_reg` occurs prior to its use in the inner loop. Initialization of `shift_reg` can significantly reduce the hardware footprint because the AOC can infer the usage of `shift_reg` in the inner loop better. The AOC does not need to carry the array data across the outer loop, hence eliminating data dependencies and hardware usage necessary for preserving the array data within the outer loop.

### Document Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
</table>
| December 2013| 13.1.1  | - Updated the section *Specify a Maximum Work-Group Size or a Required Work-Group Size.*  
- Added the section *Heterogeneous Memory Buffers.*  
- Updated the section *Single Work-Item Execution.*  
- Added the section *Performance Warning Messages.*  
- Updated the section *Single Work-Item Kernel Programming Considerations.* |
<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
</table>
| November 2013| 13.1.0  | - Reorganized information flow.  
- Updated the section *Altera SDK for OpenCL Compilation Flow*.  
- Updated the section *Pipelines*; inserted the figure *Example Multistage Pipeline Diagram*.  
- Removed the following figures:  
  - *Instruction Flow through a Five-Stage Pipeline Processor*.  
  - *Vector Addition Kernel Compiled to an FPGA*.  
  - *Effect of Kernel Vectorization on Array Summation*.  
  - *Data Flow Implementation of a Four-Element Accumulation Kernel*.  
  - *Data Flow Implementation of a Four-Element Accumulation Kernel with Loop Unrolled*.  
  - *Complete Loop Unrolling*.  
  - *Unrolling Two Loop Iterations*.  
  - *Memory Master Interconnect*.  
  - *Local Memory Read and Write Ports*.  
  - *Local Memory Configuration*.  
- Updated the section *Good Design Practices*.  
- Removed the following sections:  
  - *Predicated Execution*.  
  - *Throughput Analysis*.  
  - *Case Studies*.  
- Updated and renamed *Optimizing Data Processing Efficiency* to *Optimization of Data Processing Efficiency*.  
- Renamed *Replicating Compute Units versus Kernel SIMD Vectorization* to *Compute Unit Replication versus Kernel SIMD Vectorization*.  
- Renamed *Using num_compute_units and num_simd_work_items Together to Combination of Compute Unit Replication and Kernel SIMD Vectorization*.  
- Updated and renamed *Memory Streaming* to *Contiguous Memory Accesses*.  
- Updated and renamed *Optimizing Memory Access* to *General Guidelines on Optimizing Memory Accesses*.  
- Updated and renamed *Optimizing Memory Efficiency* to *Optimization of Memory Access Efficiency*.  
- Inserted the subsection *Single Work-Item Execution* under *Optimization of Memory Access Efficiency*. |
<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
</table>
| June 2013    | 13.0 SP1.0 | - Updated support status of OpenCL kernel source code containing complex exit paths.  
                        - Updated the figure *Effect of Kernel Vectorization on Array Summation* to correct the data flow between Store and Global Memory.  
                        - Updated content for the `unroll` pragma directive in the section *Loop Unrolling*.  
                        - Updated content of the *Local Memory* section.  
                        - Updated the figure *Local Memories Transferring Data Blocks within Matrices A and B* to correct the data transfer pattern in Matrix B.  
                        - Removed the figure *Loop Unrolling with Vectorization*.  
                        - Removed the section *Optimizing Local Memory Bandwidth*. |
| May 2013     | 13.0.1  | - Updated terminology. For example, pipeline is replaced with compute unit; vector lane is replaced with SIMD vector lane.  
                        - Added the following sections under *Good Design Practices*:  
                        - Preprocessor Macros.  
                        - Floating-Point versus Fixed-Point Representations.  
                        - Recommended Optimization Methodology.  
                        - Sequence of Optimization Techniques.  
                        - Updated code fragments.  
                        - Updated the figure *Data Flow with Multiple Compute Units*.  
                        - Updated the figure *Compute Unit Replication versus Kernel SIMD Vectorization*.  
                        - Updated the figure *Optimizing Throughput Using Compute Unit Replication and SIMD Vectorization*.  
                        - Updated the figure *Memory Streaming*.  
                        - Inserted the figure *Local Memories Transferring Data Blocks within Matrices A and B*.  
                        - Reorganized the flow of information. Number of figures, tables, and examples have been updated.  
                        - Included information on new kernel attributes: `max_share_resources` and `num_share_resources`. |
| May 2013     | 13.0.0  | - Updated pipeline discussion.  
                        - Updated case study code examples and results tables.  
                        - Updated figures. |
| November 2012| 12.1.0  | Initial release.                                                         |