Altera SDK for OpenCL

Best Practices Guide

Subscribe

Send Feedback

OCL003-15.0.0
2015.05.04

101 Innovation Drive
San Jose, CA 95134
www.altera.com

Contents

Altera SDK for OpenCL Best Practices Guide

2015.05.04

OCL003-15.0.0 Subscribe Send Feedback

The *Altera SDK for OpenCL Best Practices Guide* provides guidance on leveraging the functionalities of the Altera® Software Development Kit (SDK) for OpenCL™[1] to optimize your OpenCL[2] applications for Altera FPGAs.

This document 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. It also assumes that you have experience in creating OpenCL applications.

For more information on the OpenCL Specification version 1.0, refer to the OpenCL Reference Pages on the Khronos Group website. For detailed information on the OpenCL APIs and programming language, refer to the *OpenCL Specification version 1.0*.

Related Information

- **OpenCL Reference Pages**
- **OpenCL Specification version 1.0**

Introduction

To achieve the highest performance of your OpenCL application for FPGAs, familiarize yourself with details of the underlying hardware. In addition, 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 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

[1] The Altera SDK for OpenCL (AOCL) 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**.

[2] OpenCL and the OpenCL logo are trademarks of Apple Inc. and used by permission of the Khronos Group™.

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, heterogeneous and unique pipeline implementations. 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) ^ C) & D) >> 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.

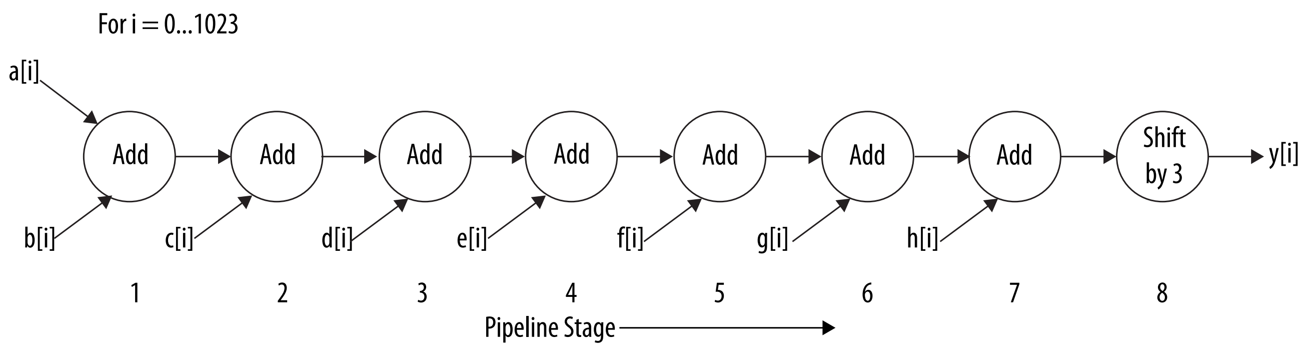
Pipelines

The designs of microprocessors, digital signal processors (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 that contributes to the final result, such as memory operation, instruction decoding, or calculation.

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

```
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



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

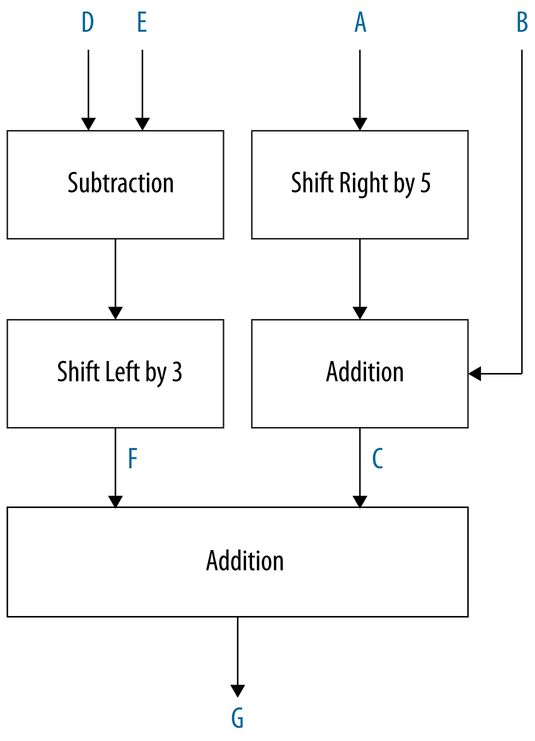
The Altera SDK for OpenCL (AOCL) pipeline does not have a set of predefined pipeline stages or instruction set. As a result, it can accommodate for the highly configurable nature of FPGAs.

Consider the following OpenCL code fragment:

```
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.

Figure 1-3: An FPGA Pipeline with Five Instructions Per Clock Cycle

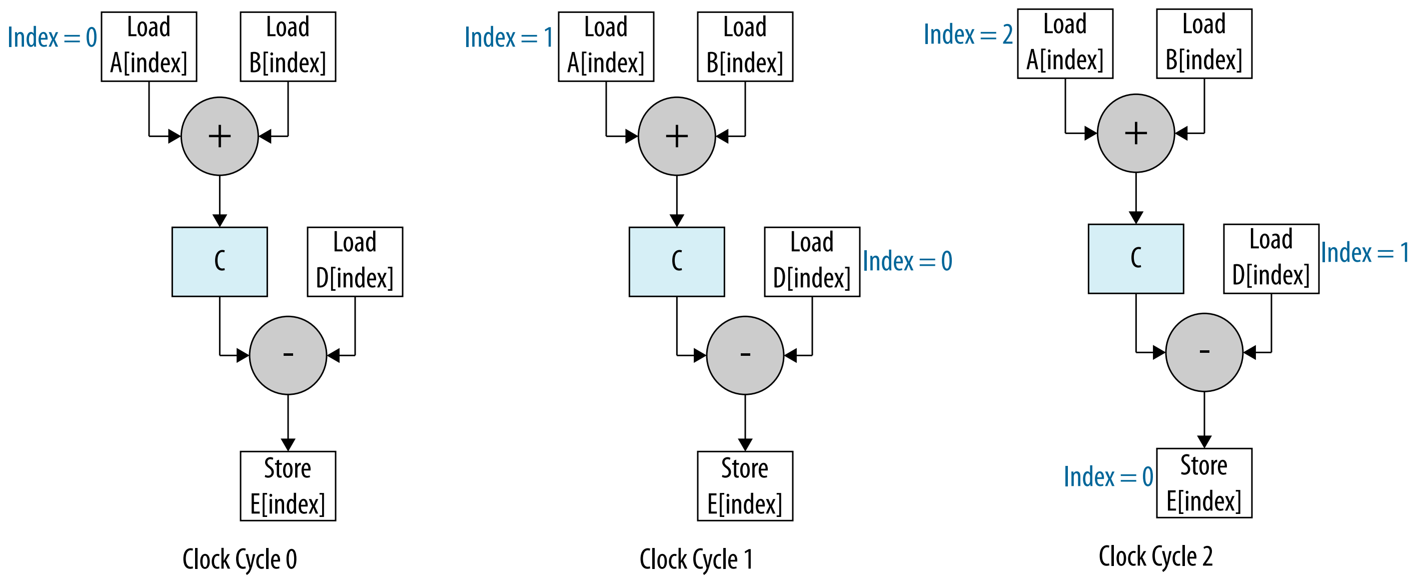
	0	1	2	3	4	5
C	(A[0] >> 5) + B[0]	(A[1] >> 5) + B[1]	(A[2] >> 5) + B[2]	(A[3] >> 5) + B[3]	(A[4] >> 5) + B[4]	(A[5] >> 5) + B[5]
F	(D[0] - E[0]) << 3	(D[1] - E[1]) << 3	(D[2] - E[2]) << 3	(D[3] - E[3]) << 3	(D[4] - E[4]) << 3	(D[5] - E[5]) << 3
G		C[0] + F[0]	C[1] + F[1]	C[2] + F[2]	C[3] + F[3]	C[4] + F[4]

Time in Clock Cycles

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:

```
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



Single Work-Item Kernel versus NDRange Kernel

In the early stages of design planning, consider whether constructing your OpenCL kernel as a single work-item might improve performance. Altera recommends that you structure your OpenCL kernel as a single work-item, if possible. However, if your kernel program does not have loop and memory dependencies, you may structure your application as an NDRange kernel because the kernel can execute multiple work-items in parallel efficiently.

Create single work-item kernels for your design if it satisfies the following criteria:

- You organize your OpenCL application in multiple kernels, use channels to transfer data among the kernels, and data processing sequence is critical to your application.

  Attention: In this scenario, if you create NDRange kernels, ensure that you understand the way the Altera SDK for OpenCL defines the order in which multiple work-items access a channel. For more information, refer to the *Multiple Work-Item Ordering* section of the *Altera SDK for OpenCL Programming Guide*.
- You cannot break down an algorithm into separate work-items easily because of data dependencies that arise when multiple work-items are in flight.

Related Information

Multiple Work-Item Ordering for Channels

Single Work-Item Execution

The Altera SDK for OpenCL (AOCL) host can execute a kernel as a single work-item, which is equivalent to launching a kernel with an NDRange size of (1, 1, 1). The OpenCL Specification verison 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 share 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 natural sequential model similar to C programming. Particularly, you do not have to partition the data across work-items.

To ensure high-throughput single work-item-based kernel execution on the FPGA, the Altera Offline Compiler (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:

```
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. An NDRange kernel requires an intricate mechanism involving local buffers and barrier constructs to pass the shifted `ai` values to each work-item. Implementing this intricate mechanism leads to suboptimal kernel performance. However, rewriting the kernel as a single work-item kernel allows the AOC to extract the parallelism between each loop iteration. 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*.

To extract the parallelism between loop iterations, optimize your kernel for loop pipelining manually. In the example above, optimize the code to enable the AOC to leverage hardware shift registers on the FPGA to generate a pipeline that shifts `ai[k]` into `ai[k-1]`. This optimization allows the loop to execute in a true pipeline fashion.

If your kernel code does not access any global or local ID by calling `get_global_id()` or `get_local_id()`, respectively, the AOC assumes your kernel is a single work-item kernel and attempts to implement loop pipelining automatically.

Limitations

The OpenCL task parallel programming model does not support the notion of a barrier in single-work-item execution. Replace barriers (`barrier`) with memory fences (`mem_fence`) in your kernel. Barriers in a pipelined loop of an OpenCL single work-item kernel cause the AOC to error out.

Good OpenCL Kernel Design Practices

With the Altera Offline Compiler (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. After you optimize this compute unit, 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.

Transfer Data Via AOCL Channels or OpenCL Pipes on page 1-7
To increase data transfer efficiency between kernels, implement the Altera SDK for OpenCL (AOCL) channels extension in your kernel programs. If you want to leverage the capabilities of channels but have the ability to run your kernel program using other SDKs, implement OpenCL pipes.

Unroll Loops on page 1-11
If your OpenCL kernel contains loop iterations, increase performance by unrolling the loop.

Optimize Floating-Point Operations on page 1-13
For floating-point operations, you can manually direct the Altera Offline Compiler (AOC) to perform optimizations that create more efficient pipeline structures in hardware and reduce the overall hardware usage.

Allocate Aligned Memory on page 1-16
Allocate host-side buffers to be at least 64-byte aligned.

Align a Struct with or without Padding on page 1-16
A properly aligned struct helps the Altera Offline Compiler (AOC) generate the most efficient hardware.

Maintain Similar Structures for Vector Type Elements on page 1-18
If you update one element of a vector type, update all the elements of the vector.

Avoid Pointer Aliasing on page 1-19
Insert the `restrict` keyword in pointer arguments whenever possible.

Avoid Expensive Functions on page 1-19
Some functions are expensive to implement in FPGAs. Expensive functions might decrease kernel performance or require a large amount of hardware to implement.

Avoid Work-Item ID-Dependent Backward Branching on page 1-20
Avoid including any work-item ID-dependent backward branching (that is, branching that occurs in a loop) in your kernel because it degrades performance.

Transfer Data Via AOCL Channels or OpenCL Pipes

To increase data transfer efficiency between kernels, implement the Altera SDK for OpenCL (AOCL) channels extension in your kernel programs. If you want to leverage the capabilities of channels but have the ability to run your kernel program using other SDKs, implement OpenCL pipes.

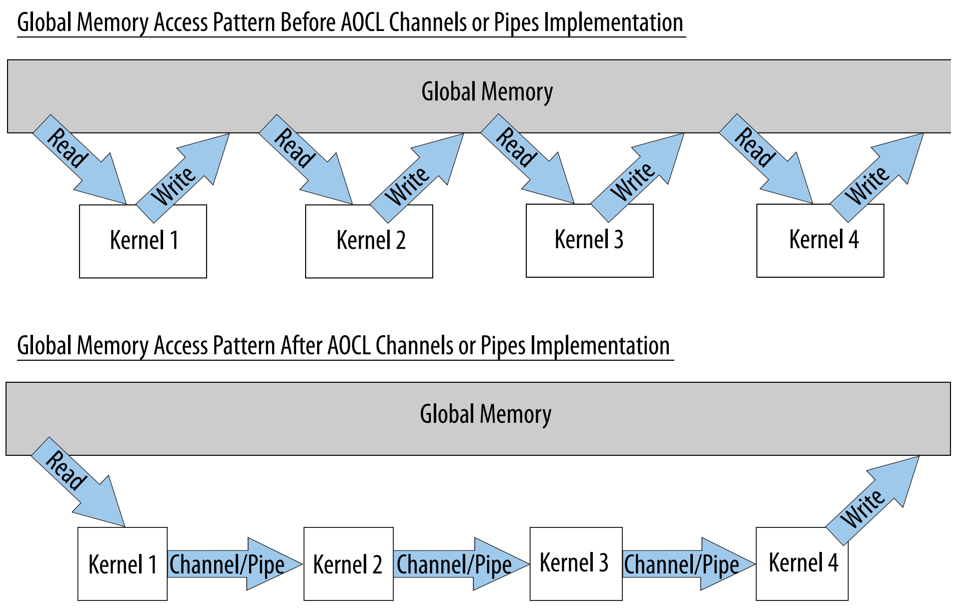
Attention: The implementation of OpenCL pipes is a beta feature for the current version of the AOCL.

Sometimes, FPGA-to-global memory bandwidth constrains the data transfer efficiency between kernels. The theoretical maximum FPGA-to-global memory bandwidth varies depending on the number of global memory banks available in the targeted Custom Platform and board. To determine the theoretical maximum bandwith for your board, refer to your board vendor's documentation.

In practice, a kernel does not achieve 100% utilization of the maximum global memory bandwidth available. The level of utilization depends on the access pattern of the algorithm.

If global memory bandwidth is a performance constraint for your OpenCL kernel, first try to break down the algorithm into multiple smaller kernels. Secondly, as shown in the figure below, eliminate some of the global memory accesses by implementing the AOCL channels or OpenCL pipes for data transfer between kernels.

Figure 1-5: Difference in Global Memory Access Pattern as a Result of Channels or Pipes Implementation


Global Memory Access Pattern Before AOCL Channels or Pipes Implementation
Global Memory
Read
Write
Kernel 1
Kernel 2
Kernel 3
Kernel 4
Global Memory Access Pattern After AOCL Channels or Pipes Implementation
Global Memory
Read
Kernel 1
Channel/Pipe
Kernel 2
Channel/Pipe
Kernel 3
Channel/Pipe
Kernel 4
Write

For more information on the usage of channels, refer to the *Implementing AOCL Channels Extension* section of the *Altera SDK for OpenCL Programming Guide*.

For more information on the usage of pipes, refer to the *Implementing OpenCL Pipes* section of the *Altera SDK for OpenCL Programming Guide*.

Related Information

- Implementing AOCL Channels Extension
- Implementing OpenCL Pipes

Channels and Pipes Characteristics

To implement channels or pipes in your OpenCL kernel program, keep in mind their respective characteristics that are specific to the Altera SDK for OpenCL (AOCL).

Default Behavior

The default behavior of channels is blocking. The default behavior of pipes is nonblocking.

Concurrent Execution of Multiple OpenCL Kernels

You can execute multiple OpenCL kernels concurrently. To enable concurrent execution, modify the host code to instantiate multiple command queues. Each concurrently executing kernel is associated with a separate command queue.

Important: Pipe-specific considerations:

The OpenCL pipe modifications outlined in *Ensuring Compatibility with Other OpenCL SDKs* in the *Altera SDK for OpenCL Programming Guide* allow you to run your kernel on the AOCL. However, they do not maximize the kernel throughput. The OpenCL Specification version 2.0 requires that pipe writes occur before pipe reads so that the kernel is not reading from an empty pipe. As a result, the kernels cannot execute concurrently. Because the AOCL supports concurrent execution, you can modify your host application and kernel program to take advantage of this capability. The modifications increase the throughput of your application; however, you can no longer port your kernel to another SDK. Despite this limitation, the modifications are minimal, and it does not require much effort to maintain both types of code.

To enable concurrent execution of kernels containing pipes, replace the `depth` attribute in your kernel code with the `blocking` attribute (that is, `__attribute__((blocking))`). The `blocking` attribute introduces a blocking behavior in the `read_pipe` and `write_pipe` function calls. The call site blocks kernel execution until the other end of the pipe is ready.

If you add both the `blocking` attribute and the `depth` attribute to your kernel, the `read_pipe` calls will only block when the pipe is full, and the `write_pipe` calls will only block when the pipe is empty. Blocking behavior causes an implicit synchronization between the kernels, which forces the kernels to run in lock step with each other.

Implicit Kernel Synchronization

Synchronize the kernels implicitly via blocking channel calls or blocking pipe calls. Consider the following examples:

Table 1-1: Blocking Channel and Pipe Calls for Kernel Synchronization

Kernel with Blocking Channel Call	Kernel with Blocking Pipe Call

`#pragma OPENCL EXTENSION`
`    cl_altera_channels : enable`
 	
`channel int c0;`
 	
`__kernel`
`void producer (__global int * in_buf)`
`{`
`  for (int i=0; i<10; i++)`
`  {`
`    write_channel_altera(c0, in_buf[i]);`
`  }`
`}`
 	
`__kernel`
`void consumer (__global int * ret_buf)`
`{`
`  for(int i=0; i<10; i++)`
`  {`
`    ret_buf[i]=read_channel_altera(c0);`
`  }`
`}`
	`__kernel`
	`void producer (__global int * in_buf,`
	`  write_only pipe int __attribute__`
	`  ((blocking)) c0)`
	`{`
	`  for (int i=0;i<10; i++)`
	`  {`
	`    write_pipe(c0, &in_buf[i]);`
	`  }`
	`}`
	
	`__kernel`
	`void consumer (__global int * ret_buf,`
	`  read_only pipe int __attribute__`
	`  ((blocking)) c0)`
	`{`
	`  for (int i=0; i<10; i++)`
	`  {`
	`    int x;`
	`    read_pipe(c0, &x);`
	`    ret_buf[i]=x;`
	`  }`
	`}`

You can synchronize the kernels such that a `producer` kernel writes data and a `consumer` kernel reads the data during each loop iteration. If the `write_channel_altera` or `write_pipe` call in `producer` does not write any data, `consumer` blocks and waits at the `read_channel_altera` or `read_pipe` call until `producer` sends valid data, and vice versa.

Data Persistence Across Invocations

After the `write_channel_altera` call writes data to a channel or the `write_pipe` call writes data to a pipe, the data is persistent across work-groups and NDRange invocations. Data that a work-item writes to a channel or a pipe remains in that channel or pipe until another work-item reads from it. In addition, the order of data in a channel or a pipe is equivalent to the sequence of write operations to that channel or pipe, and the order is independent of the work-item that performs the write operation.

For example, if multiple work-items try to access a channel or a pipe simultaneously, only one work-item can access it. The `write_channel_altera` call or `write_pipe` call writes the particular work-item data, called *DATAX*, to the channel or pipe, respectively. Similarly, the first work-item to access the channel or pipe reads DATAX from it. This sequential order of read and write operations makes channels and pipes an effective way to share data between kernels.

Imposed Work-Item Order

The AOCL imposes a work-item order to maintain the consistency of the read and write operations for a channel or a pipe.

Related Information

Ensuring Compatibility with Other OpenCL SDKs

Execution Order for Channels and Pipes

Each channel or pipe call in a kernel program translates into an instruction executed in the FPGA pipeline. The execution of a channel call or a pipe call occurs if a valid work-item executes through the pipeline. However, even if there is no control or data dependence between channel or pipe calls, their execution might not achieve perfect instruction-level parallelism in the kernel pipeline.

Consider the following code examples:

Table 1-2: Kernel with Two Read Channel or Pipe Calls

Kernel with Two Read Channel Calls	Kernel with Two Read Pipe Calls

`__kernel void`
`consumer ( __global uint*restrict dst)`
`{`
`  for(int i=0; i<5; i++)`
`  {`
`    dst[2*i]=read_channel_altera(c0);`
`    dst[2*i+2]=read_channel_altera(c1);`
`  }`
`}`
	`__kernel void`
	`consumer (__global uint*restrict dst,`
	`  read_only pipe uint`
	`    __attribute__((blocking)) c0,`
	`  read_only pipe uint`
	`    __attribute__((blocking)) c1)`
	`{`
	`  for (int i=0;i<5;i++)`
	`  {`
	`    read_pipe(c0, &dst[2*i]);`
	`    read_pipe(c1, &dst[2*i+2]);`
	`  }`
	`}`

The code example on the left makes two read channel calls. The code example on the right makes two read pipe calls. In most cases, the kernel executes these channel or pipe calls in parallel; however, channel and pipe call executions might occur out of sequence. Out-of-sequence execution means that the read operation from `c1` can occur and complete before the read operation from `c0`.

Optimize Buffer Inference for Channels or Pipes

In addition to the manual addition of buffered channels or pipes, the Altera Offline Compiler (AOC) improves kernel throughput by adjusting buffer sizes whenever possible.

During compilation, the AOC computes scheduling mismatches between interacting channels or pipes. These mismatches might cause imbalances between read and write operations. The AOC performs buffer inference optimization automatically to correct the imbalance.

Consider the following examples:

Table 1-3: Buffer Inference Optimization for Channels and Pipes

Kernel with Channels

```
__kernel void producer (
  __global const uint * restrict src,
  const uint iterations)
{
  for(int i=0; i < iteration; i++)
  {
    write_channel_altera(c0,src[2*i]);
    write_channel_altera(c1,src[2*i+1]);
  }
}

__kernel void consumer (
  __global uint * restrict dst,
  const uint iterations)
{
  for(int i=0; i < iterations; i++)
  {
    dst[2*i]=read_channel_altera(c0);
    dst[2*i+1]=read_channel_altera(c1);
  }
}
```

Kernel with Pipes

```
__kernel void producer (
  __global const uint * restrict src,
  const uint iterations,
  write_only pipe uint
    __attribute__((blocking)) c0,
  write_only pipe uint
    __attribute__((blocking)) c1)
{
  for(int i=0; i < iteration; i++)
  {
    write_pipe(c0,&src[2*i]);
    write_pipe(c1,&src[2*i+1]);
  }
}

__kernel void consumer (
  __global uint * restrict dst,
  const uint iterations,
  read_only pipe uint
    __attribute__((blocking)) c0,
  read_only pipe uint
    __attribute__((blocking)) c1)
{
  for(int i=0; i < iterations; i++)
  {
    read_pipe(c0,&dst[2*i]);
    read_pipe(c1,&dst[2*i+1]);
  }
}
```

The AOC performs buffer inference optimization if channels or pipes between kernels cannot form a cycle. A *cycle* between kernels is a path that originates from a kernel, through a write channel or a write pipe call, and returns to the original kernel. For the example, assume that the write channel or write pipe calls in the kernel `producer` are scheduled 10 cycles apart and the read channel or read pipe calls are scheduled 15 cycles apart. There exists a temporary mismatch in the read and write operations to `c1` because five extra write operations might occur before a read operation to `c1` occurs. To correct this imbalance, the AOC assigns a buffer size of five cycles to `c1` to avoid stalls. The extra buffer capacity decouples the `c1` write operations in the `producer` kernel and the `c1` read operations in the `consumer` kernel.

Unroll Loops

You can control the way the Altera Offline Compiler (AOC) translates OpenCL kernel descriptions to hardware resources. If your OpenCL kernel contains loop iterations, increase performance by unrolling the loop. Loop unrolling decreases the number of iterations that 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:

```
__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 from the end of the pipeline 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, add a `#pragma unroll` directive to the main loop, as shown in the code example below. Keep in mind loop unrolling significantly changes the structure of the compute unit that the AOC creates.

```
__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 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.

Caution: 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 from global memory 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. Consequently, the hardware implementation of your kernel might not fit into the FPGA.
- You specify complete unrolling and 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, specify the `#pragma unroll <N>` directive, where *<N>* is the unroll factor. The unroll factor limits the number of iterations that the AOC unrolls. For example, to prevent a loop in your kernel from unrolling, add the directive `#pragma unroll 1` to that loop.

Refer to *Good Design Practices for Single Work-Item Kernel* for tips on constructing well-structured loops.

Related Information

Good Design Practices for Single Work-Item Kernel on page 1-50

Optimize Floating-Point Operations

For floating-point operations, you can manually direct the Altera Offline Compiler (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.

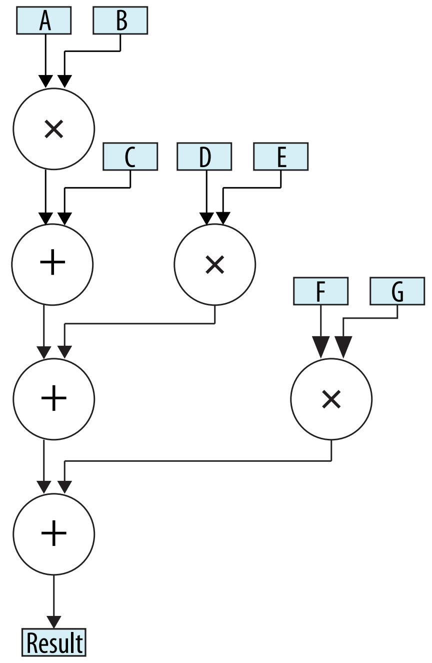
Tree Balancing

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

```
result = (((A * B) + C) + (D * E)) + (F * G);
```

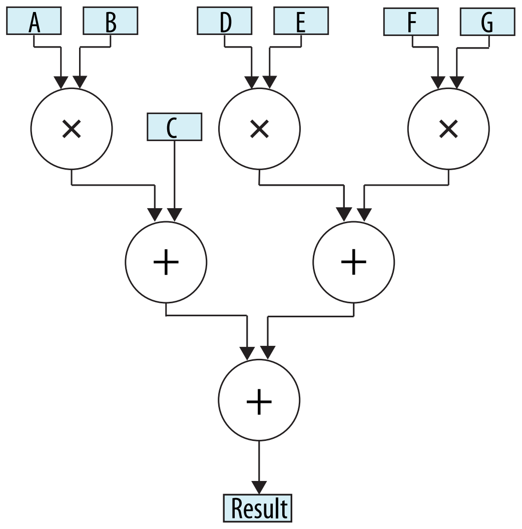
By default, the AOC creates an implementation that resembles a long vine for such computations:

Figure 1-6: Default Floating-Point Implementation



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

Figure 1-7: 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 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 <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` option of the `aoc` command. If your program can tolerate small differences in floating-point results, invoke the following command:

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

The `--fpc` option 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 performs 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-accumulate (FMAC) instruction available in new processor architectures. The AOC can provide fused floating-point mathematical capabilities for many combinations of floating-point operators in your kernel.

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 Altera Offline Compiler (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:

```
__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.

Allocate Aligned Memory

Allocate host-side buffers to be at least 64-byte aligned. Allocating the host-side buffers 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:

  ```
  #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:

  ```
  #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);`

Align a Struct with or without Padding

A properly aligned struct helps the Altera Offline Compiler (AOC) generate the most efficient hardware. A proper struct alignment means that the alignment can be evenly divided by the struct size. Hardware efficiency increases with increasing alignment.

The AOC conforms with the ISO C standard which requires the alignment of a struct to satisfy all of the following criteria:

1. The alignment must be an integer multiple of the lowest common multiple between the alignments of all struct members.
2. The alignment must be a power of two.

You may set the struct alignment by including the `aligned(N)` attribute in your kernel code. Without an aligned attribute, the AOC determines the alignment of each struct in an array of struct based on the size of the struct. Consider the following example:

```
__kernel void test (struct mystruct* A,
                    struct mystruct* B)
{
    A[get_global_id(0)] = B[get_global_id(0)];
}
```

If the size of `mystruct` is 101 bytes, each load or store access will be 1-byte aligned. If the size of `mystruct` is 128 bytes, each load or store access will be 128-byte aligned, which generates the most efficient hardware.

When the struct fields are not aligned within the struct, the AOC inserts padding to align them. Inserting padding between struct fields affects hardware efficiency in the following manner:

1. Increases the size of the struct
2. Might affect the alignment

To prevent the AOC from inserting padding, include the `packed` attribute in your kernel code. The aforementioned ISO C standard applies when determining the alignment of a packed or unpacked struct. Consider the following example:

```
struct mystruct1
{
    char a;
    int b;
};
```

The size of `mystruct1` is 8 bytes. Therefore, the struct is 8-byte aligned, resulting in efficient accesses in the kernel. Now consider another example:

```
struct mystruct2
{
    char a;
    int b;
    int c;
};
```

The size of `mystruct2` is 12 bytes and the struct is 4-byte aligned. Because the struct fields are padded and the struct is unaligned, accesses in the kernel are inefficient.

Below is an example of a struct that includes the `packed` attribute:

```
__attribute__((packed))
struct mystruct3
{
    char a;
    int b;
    int c;
};
```

The size of `mystruct3` is 9 bytes; therefore, the struct is 1-byte aligned. Because there is no padding between struct fields, accesses in this kernel are more efficient than accesses in `mystruct2`. However, `mystruct3` is unaligned.

Below is an example of a struct that is aligned and is not padded:

```
__attribute__((packed))
struct mystruct4
{
    char a;
    int b;
    int c;
    char d[7];
};
```

The size of `mystruct4` is 16 bytes. Because `mystruct4` is aligned and there is no padding between struct fields, accesses in this kernel are more efficient than accesses in `mystruct3`.

To include both the `aligned(N)` and `packed` attributes in a struct, consider the following example:

```
__attribute__((packed))
__attribute__((aligned(16)))
struct mystruct5
{
    char a;
    int b;
    int c;
};
```

The size of `mystruct5` is 9 bytes. Because of the `aligned(16)` attribute, the struct is stored at 16-byte aligned addresses in an array. Because `mystruct5` is 16-byte aligned and has no padding, accesses in this kernel will be efficient.

For more information on struct alignment and the `aligned(N)` and `packed` attributes, refer to the following documents:

- Section 6.11.1 of the *OpenCL Specification version 1.2*
- *Disabling Insertion of Data Structure Padding* section of the *Altera SDK for OpenCL Programming Guide*
- *Specifying the Alignment of a Struct* section of the *Altera SDK for OpenCL Programming Guide*

Related Information

- **OpenCL Specification version 1.2**
- **Disabling Insertion of Data Structure Padding**
- **Specifying the Alignment of a Struct**

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:

```
__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.
}
```

Avoid Pointer Aliasing

Insert the `restrict` keyword in pointer arguments whenever possible. Including the `restrict` keyword in pointer arguments prevents the Altera Offline Compiler (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:

```
__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 incorrect results.

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, absolute value, and comparison

  Note: For more information on optimizing floating-point operations, refer to the *Optimize 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 a 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:

```
__kernel void myKernel (__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);
}
```

The result of this calculation is always the same. To avoid this redundant and hardware resource-intensive operation, perform the calculation in the host application and then pass the result to the kernel as an argument for all work-items in the NDRange to use. The modified code is shown below:

```
__kernel void myKernel (__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 Altera Offline Compiler (AOC) consolidates operations that are not work-item-dependent across the entire NDRange into a single operation. It 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

Optimize Floating-Point Operations on page 1-13

Avoid Work-Item ID-Dependent Backward Branching

The Altera Offline Compiler (AOC) collapses conditional statements into single bits that indicate when 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 kernels that include forward branches, such as conditional statements, efficiently.

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`:

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

Profile Your Kernel to Identify Performance Bottlenecks

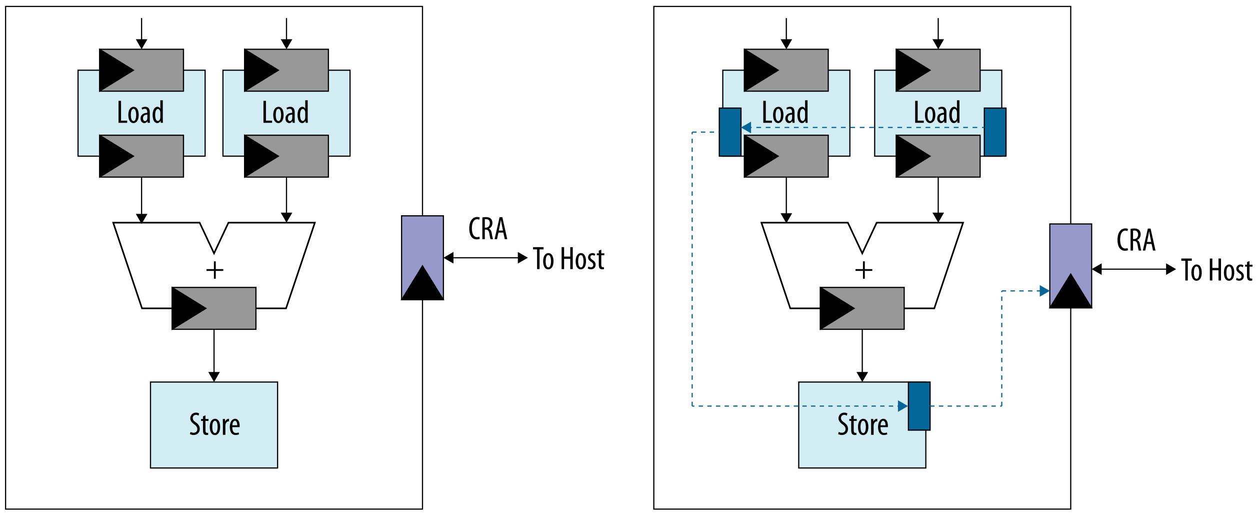
The Altera SDK for OpenCL Profiler generates data that helps you assess OpenCL kernel performance. The Profiler instruments the kernel pipeline with performance counters. These counters collect kernel performance data, which you can review via the profiler GUI.

Consider the following OpenCL kernel program:

```
__kernel void add (__global int * a,
                   __global int * b,
                   __global int * c)
{
    int gid = get_global_id(0);
    c[gid] = a[gid]+b[gid];
}
```

As shown in the figure below, the profiler instruments and connects performance counters in a daisy chain throughout the pipeline generated for the kernel program. The host then reads the data collected by these counters. For example, in PCI Express® (PCIe®)-based systems, the host reads the data via the PCIe control register access (CRA) or control and status register (CSR) port.

Figure 1-8: AOCL Profiler: Performance Counters Instrumentation



Work-item execution stalls might occur at various stages of an AOCL pipeline. Applications with large amounts of memory accesses or load and store operations might stall frequently to enable the completion of memory transfers. The Profiler helps identify the load and store operations that cause the majority of stalls within a kernel pipeline.

For usage information on the AOCL Profiler, refer to the *Profiling Your OpenCL Kernel* section of the *Altera SDK for OpenCL Programming Guide.*

Related Information

Profiling Your OpenCL Kernel

AOCL Profiler GUI

The Altera SDK for OpenCL (AOCL) Profiler GUI displays statistical information collected from memory and channel or pipe accesses.

Table 1-4: Summary Heading in the AOCL Profiler GUI

Heading	Description
Board	Name of the accelerator board that the Altera Offline Compiler (AOC) uses during kernel emulation and execution.
Global Memory BW (DDR)	Maximum theoretical global memory bandwidth available for each memory type (for example, DDR).

Directly below the summary heading, you can view detailed profile information by clicking on the available tabs.

Important: In the following sections, information that relates to the AOCL channels also applies to OpenCL pipes.

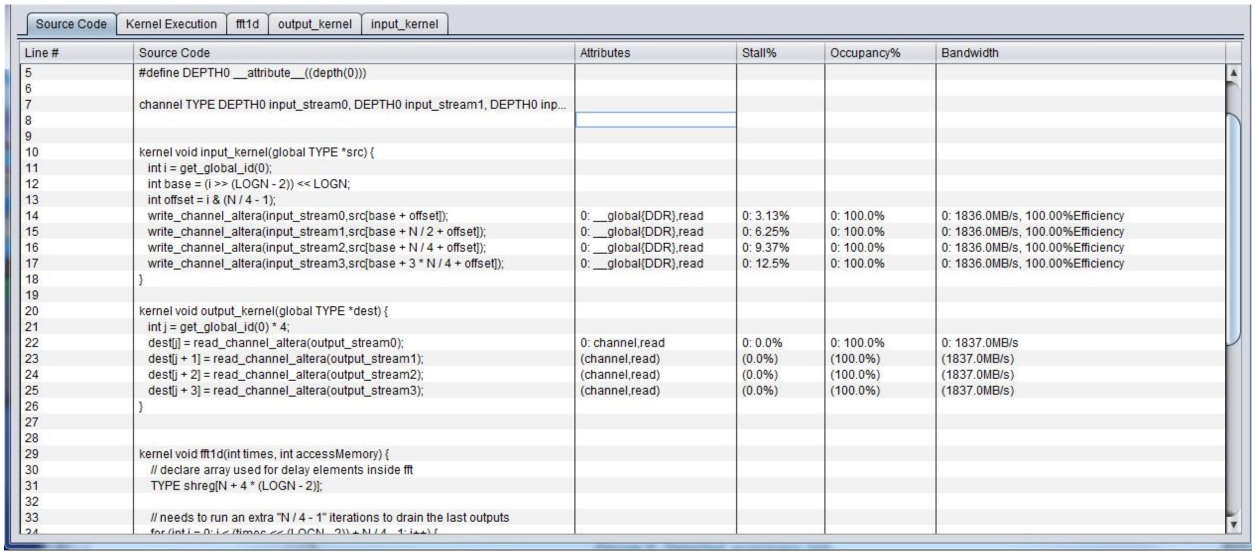
Source Code Tab on page 1-22
The **Source Code** tab in the Altera SDK for OpenCL (AOCL) Profiler GUI contains source code information and detailed statistics about memory and channel accesses.

Kernel Execution Tab on page 1-26
The **Kernel Execution** tab in the Altera SDK for OpenCL (AOCL) Profiler GUI provides a graphical representation of the overall kernel program execution process.

Source Code Tab

The **Source Code** tab in the Altera SDK for OpenCL (AOCL) Profiler GUI contains source code information and detailed statistics about memory and channel accesses.

Figure 1-9: The Source Code tab in the AOCL Profiler GUI



Line #	Source Code	Attributes	Stall%	Occupancy%	Bandwidth
5	#define DEPTH0 __attribute__((depth(0)))				
6					
7	channel TYPE DEPTH0 input_stream0, DEPTH0 input_stream1, DEPTH0 inp...				
8					
9					
10	kernel void input_kernel(global TYPE *src) {				
11	int i = get_global_id(0);				
12	int base = (i >> (LOGN - 2)) << LOGN;				
13	int offset = i & (N / 4 - 1);				
14	write_channel_altera(input_stream0,src[base + offset]);	0: __global{DDR},read	0: 3.13%	0: 100.0%	0: 1836.0MB/s, 100.00%Efficiency
15	write_channel_altera(input_stream1,src[base + N / 2 + offset]);	0: __global{DDR},read	0: 6.25%	0: 100.0%	0: 1836.0MB/s, 100.00%Efficiency
16	write_channel_altera(input_stream2,src[base + N / 4 + offset]);	0: __global{DDR},read	0: 9.37%	0: 100.0%	0: 1836.0MB/s, 100.00%Efficiency
17	write_channel_altera(input_stream3,src[base + 3 * N / 4 + offset]);	0: __global{DDR},read	0: 12.5%	0: 100.0%	0: 1836.0MB/s, 100.00%Efficiency
18	}				
19					
20	kernel void output_kernel(global TYPE *dest) {				
21	int j = get_global_id(0) * 4;				
22	dest[j] = read_channel_altera(output_stream0);	0: channel,read	0: 0.0%	0: 100.0%	0: 1837.0MB/s
23	dest[j + 1] = read_channel_altera(output_stream1);	(channel,read)	(0.0%)	(100.0%)	(1837.0MB/s)
24	dest[j + 2] = read_channel_altera(output_stream2);	(channel,read)	(0.0%)	(100.0%)	(1837.0MB/s)
25	dest[j + 3] = read_channel_altera(output_stream3);	(channel,read)	(0.0%)	(100.0%)	(1837.0MB/s)
26	}				
27					
28					
29	kernel void fft1d(int times, int accessMemory) {				
30	// declare array used for delay elements inside fft				
31	TYPE shreg[N + 4 * (LOGN - 2)];				
32					
33	// needs to run an extra "N / 4 - 1" iterations to drain the last outputs				

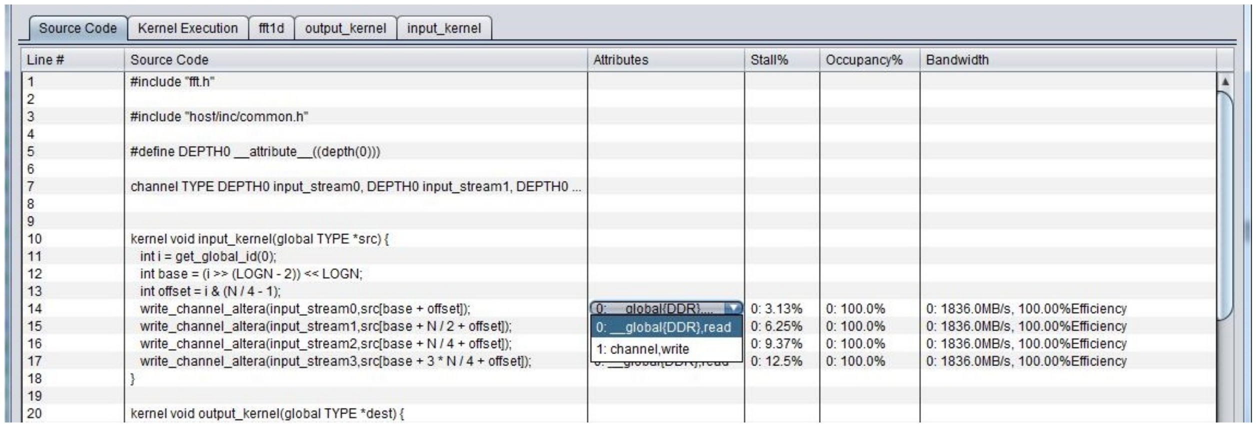
The **Source Code** tab provides detailed information on specific lines of kernel code.

Table 1-5: Types of Information Available in the Source Code Tab

Column	Description	Access Type
Attributes	Memory or channel attributes information such as memory type (local or global), corresponding memory system (DDR or quad data rate (QDR)), and read or write access.	All memory and channel accesses
Stall%	Percentage of time the memory or channel access is causing pipeline stalls.	All memory and channel accesses
Occupancy%	Percentage of the overall profiled time frame when a valid work-item executes the memory or channel instruction.	All memory and channel accesses
Bandwidth	Average memory bandwidth that the memory access uses and its overall efficiency. For each global memory access, FPGA resources are assigned to acquire data from the global memory system. However, the amount of data a kernel program uses might be less than the acquired data. The overall efficiency is the percentage of total bytes, acquired from the global memory system, that the kernel program uses.	Global memory accesses

If a line of source code instructs more than one memory or channel operations, the profile statistics appear in a drop-down list box and you may select to view the relevant information.

Figure 1-10: Source Code Tab: Drop-Down List for Multiple Memory or Channel Operations

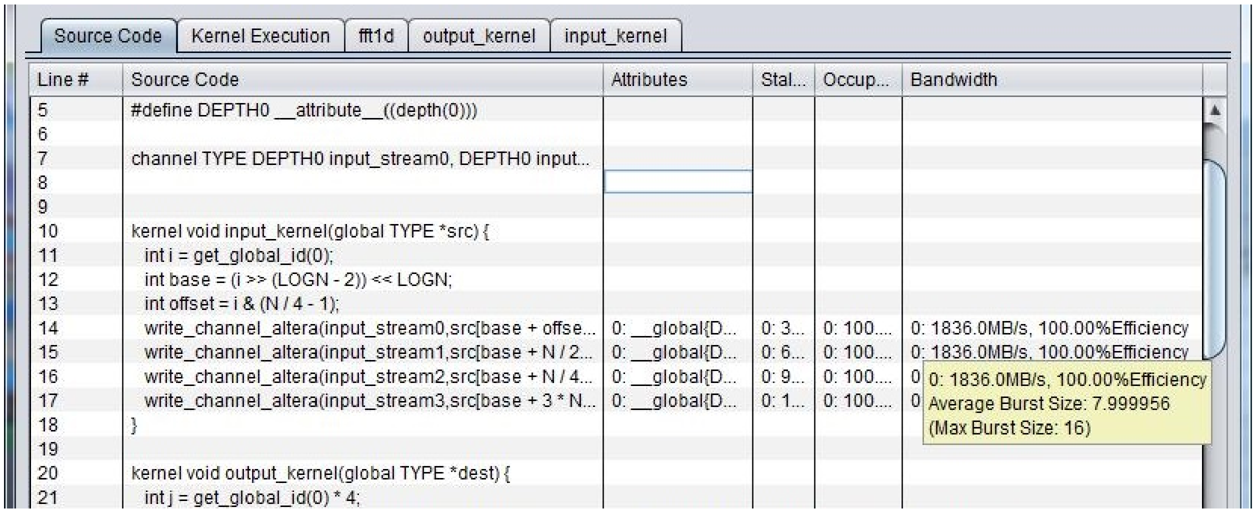


Line #	Source Code	Attributes	Stall%	Occupancy%	Bandwidth
1	#include "fft.h"				
2					
3	#include "host/inc/common.h"				
4					
5	#define DEPTH0 __attribute__((depth(0)))				
6					
7	channel TYPE DEPTH0 input_stream0, DEPTH0 input_stream1, DEPTH0 ...				
8					
9					
10	kernel void input_kernel(global TYPE *src) {				
11	int i = get_global_id(0);				
12	int base = (i >> (LOGN - 2)) << LOGN;				
13	int offset = i & (N / 4 - 1);				
14	write_channel_altera(input_stream0,src[base + offset]);	0: __global{DDR}...	0: 3.13%	0: 100.0%	0: 1836.0MB/s, 100.00%Efficiency
15	write_channel_altera(input_stream1,src[base + N / 2 + offset]);	0: __global{DDR},read	0: 6.25%	0: 100.0%	0: 1836.0MB/s, 100.00%Efficiency
16	write_channel_altera(input_stream2,src[base + N / 4 + offset]);	1: channel,write	0: 9.37%	0: 100.0%	0: 1836.0MB/s, 100.00%Efficiency
17	write_channel_altera(input_stream3,src[base + 3 * N / 4 + offset]);	0: __global{DDR},read	0: 12.5%	0: 100.0%	0: 1836.0MB/s, 100.00%Efficiency
18	}				
19					
20	kernel void output_kernel(global TYPE *dest) {				

Tool Tip Options

To obtain additional information about the kernel source code, hover your mouse over channel accesses in the code to activate the tool tip.

Figure 1-11: The AOCL Profiler GUI: Source Code Tab Tool Tip



Line #	Source Code	Attributes	Stal...	Occup...	Bandwidth
5	#define DEPTH0 __attribute__((depth(0)))				
6					
7	channel TYPE DEPTH0 input_stream0, DEPTH0 input...				
8					
9					
10	kernel void input_kernel(global TYPE *src) {				
11	int i = get_global_id(0);				
12	int base = (i >> (LOGN - 2)) << LOGN;				
13	int offset = i & (N / 4 - 1);				
14	write_channel_altera(input_stream0,src[base + offse...	0: __global{D...	0: 3...	0: 100....	0: 1836.0MB/s, 100.00%Efficiency
15	write_channel_altera(input_stream1,src[base + N / 2...	0: __global{D...	0: 6...	0: 100....	0: 1836.0MB/s, 100.00%Efficiency
16	write_channel_altera(input_stream2,src[base + N / 4...	0: __global{D...	0: 9...	0: 100....	0
17	write_channel_altera(input_stream3,src[base + 3 * N...	0: __global{D...	0: 1...	0: 100....	0
18	}				
19					
20	kernel void output_kernel(global TYPE *dest) {				
21	int j = get_global_id(0) * 4;				

```
0: 1836.0MB/s, 100.00%Efficiency
Average Burst Size: 7.999956
(Max Burst Size: 16)
```

Attention: If your kernel undergoes memory optimization that consolidates hardware resources that implement multiple memory operations, statistical data might not be available for each memory operation. One set of statistical data will map to the point of consolidation in hardware.

Table 1-6: Types of Information that a Source Code Tab Tool Tip Might Provide

Column	Tool Tip	Description	Example Message	Access Type
Attributes	Cache Hits	The number of memory accesses using the cache. A high cache hit rate reduces memory bandwidth utilization.	Cache Hit%=30%	Global memory
	Unaligned Access	The percentage of unaligned memory accesses. A high unaligned access percentage signifies inefficient memory accesses. Consider modifying the access patterns in the kernel code to improve efficiency.	Unaligned Access %=20%	Global memory
	Statically Coalesced	Indication of whether the load or store memory operation is statically coalesced. Generally, static memory coalescing merges multiple memory accesses that access consecutive memory addresses into a single wide access.	Coalesced	Global or local memory
Occupancy%	Activity	The percentage of time a predicated channel or memory instruction is enabled (that is, when conditional execution is true). **Note:** The activity percentage might be less than the occupancy of the instruction.	Activity=20%	Global or local memory, and channels

Column	Tool Tip	Description	Example Message	Access Type
Bandwidth	Burst Size	The average burst size of the memory operation. If the memory system does not support burst mode (for example, on-chip RAM), no burst information will be available.	Average Burst Size=7.6 (Max Burst=16)	Global memory

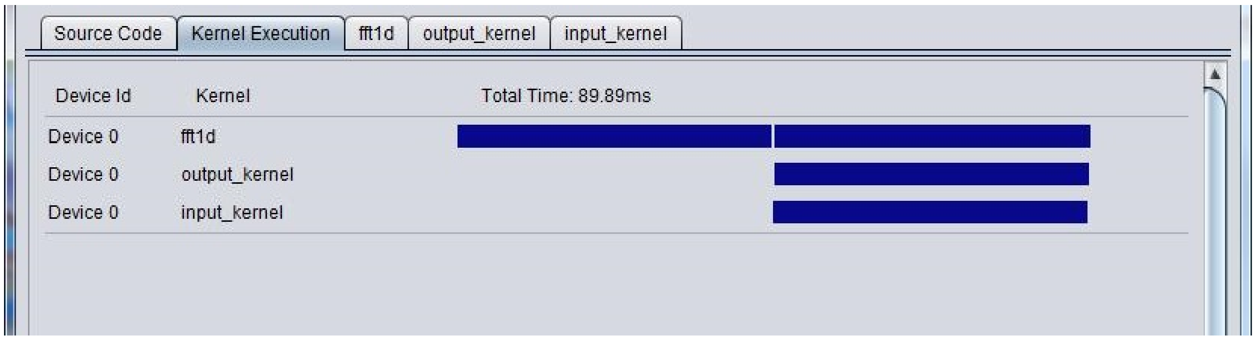
Kernel Execution Tab

The **Kernel Execution** tab in the Altera SDK for OpenCL (AOCL) Profiler GUI provides a graphical representation of the overall kernel program execution process.
It illustrates the execution time of each kernel and provides insight into the interactions between different kernel executions.

For example, if you run the host application from a networked directory with slow network disk accesses, the GUI can display the resulting delays between kernel launches while the runtime stores profile output data to disk.

Attention: To avoid potential delays between kernel executions and increases in the overall execution time of the host application, run your host application from a local disk.

Figure 1-12: The Kernel Execution Tab in the Profiler GUI



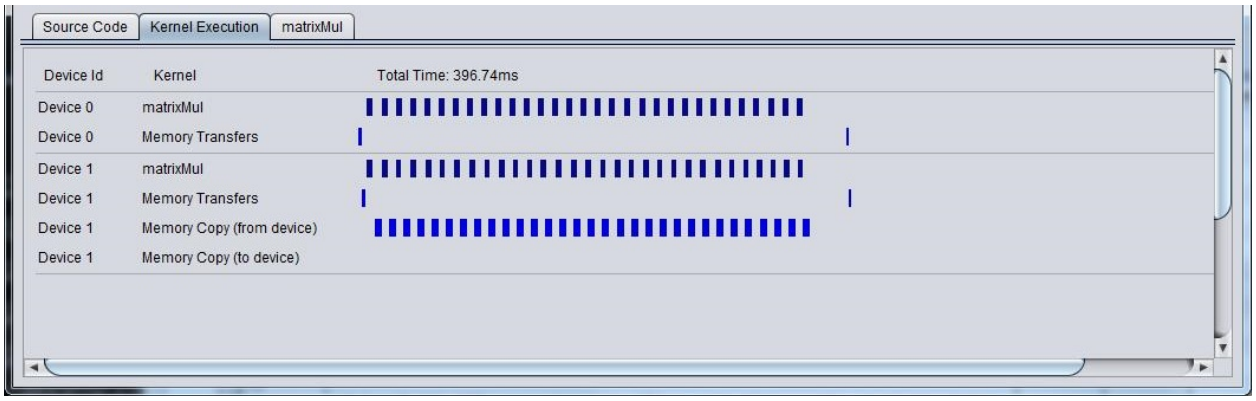
Device Id	Kernel	Total Time: 89.89ms
Device 0	fft1d	
Device 0	output_kernel	
Device 0	input_kernel	

The horizontal bar graph represents kernel execution through time. The combination of the two bars shown in the first entry (fft1d) represents the total time. The second and last entries show kernel executions that occupy the time span. These bars represent the concurrent execution of `output_kernel` and `input_kernel`, and indicate that the kernels share common resources such as memory bandwidth.

Tip: You can examine profile data for specific execution times. In the example above, when you double-click the bar on the left for fft1d, another window opens to display profile data for that specific kernel execution event.

The **Kernel Execution** tab also displays information on memory transfers between the host and your devices, shown below:

Figure 1-13: Kernel Execution Tab: Host-Device Memory Transfer Information



Device Id	Kernel	Total Time: 396.74ms
Device 0	matrixMul	
Device 0	Memory Transfers	
Device 1	matrixMul	
Device 1	Memory Transfers	
Device 1	Memory Copy (from device)	
Device 1	Memory Copy (to device)	

To enable the display of memory transfer information, set the environment variable *ACL_PROFILE_TIMER* to a value of 1 and then run your host application. Setting the *ACL_PROFILE_TIMER* environment variable enables the recording of memory transfers. The information is stored in the **profile.mon** file and is then parsed by the profiler GUI.

Interpreting the Profiling Information

The profiling information helps you identify poor memory or channel behaviors that lead to unsatisfactory kernel performance.

Below are examples of how you can interpret the statistics that the profiler reports.

Important: In the following sections, information that relates to the AOCL channels also applies to OpenCL pipes.

Occupancy on page 1-27
Occupancy measures the percentage of time a new work-item enters a memory instruction.

High Stall Percentage on page 1-29
A high stall percentage implies that the memory or channel instruction is unable to fulfill the access request because of contention for memory bandwidth or channel buffer space.

Low Bandwidth Efficiency on page 1-29
Low bandwidth efficiency occurs when excessive amount of bandwidth is necessary to obtain useful data.

Stalling Channels on page 1-29
If an I/O channel stalls, it implies that the I/O channel cannot keep up with the kernel.

Occupancy

Occupancy measures the percentage of time a new work-item enters a memory instruction.

The Altera Offline Compiler (AOC) generates a pipeline architecture where work-items traverse through the pipeline stages sequentially (that is, in a pipeline-parallel manner). As soon as a pipeline stage becomes empty, a work-item enters and occupies the stage. Pipeline parallelism also applies to iterations of pipelined loops, where iterations enter a pipelined loop sequentially.

- The profiler reports a high occupancy percentage if the AOC generates a highly efficient pipeline from your kernel, where work-items or iterations are moving through the pipeline stages without stalling.
- Occupancy percentage decreases with increasing amount of stalls in the pipeline.
    - If work-items cannot enter the pipeline consecutively, they insert bubbles into the pipeline.
    - In loop pipelining, loop-carried dependencies also form bubbles in the pipeline because of bubbles that exist between iterations.

Related Information

Source Code Tab on page 1-22

Low Occupancy Percentage

A low occupancy percentage implies that a work-item is accessing the load and store operations or the channel infrequently. This behavior is expected for load and store operations or channels that are in non-critical loops. However, if the memory or channel instruction is in critical portions of the kernel code and the occupancy or activity percentage is low, it implies that a performance bottleneck exists because work-items or loop iterations are not being issued in the hardware.

Consider the following code example:

```
__kernel void proc (__global int * a, ...)
{
    for (int i=0; i < N; i++)
    {
        for (int j=0; j < 1000; j++)
        {
            write_channel_altera (c0, data0);
        }
        for (int k=0; k < 3; k++)
        {
            write_channel_altera (c1, data1);
        }
    }
}
```

Assuming all the loops are pipelined, the first inner loop with a trip count of 1000 is the critical loop. The second inner loop with a trip count of three will be executed infrequently. As a result, you can expect that the occupancy and activity percentages for channel `c0` are high and for channel `c1` are low.

Also, occupancy percentage might be low if you define a small work-group size, the kernel might not receive sufficient work-items. This is problematic because the pipeline is empty generally for the duration of kernel execution, which leads to poor performance.

Occupancy versus Activity

Activity measures the percentage of time that a predicated instruction is enabled. The primary difference between occupancy and activity relates to *predication*.

A work-item or loop iteration can occupy a memory instruction even if it is predicated. If the branch statements do not contain loops, the Altera Offline Compiler (AOC) converts the branches to minimize control flow, which leads to more efficient hardware. As part of the conversion, memory and channel instructions must be predicated and the output results much be selected through multiplexer logic.

Activity percentages available in the tool tips do not account for predicated accesses. Therefore, you can identify predicated instructions based on low activity percentages. Despite having low activity percentages, these instructions might have high occupancies.

Related Information

Tool Tip Options on page 1-23

High Stall Percentage

A high stall percentage implies that the memory or channel instruction is unable to fulfill the access request because of contention for memory bandwidth or channel buffer space.

Memory instructions stall often whenever bandwidth usage is inefficient or if a large amount of data transfer is necessary during the execution of your application. Inefficient memory accesses lead to suboptimal bandwidth utilization. In such cases, analyze your kernel memory accesses for possible improvements.

Channel instructions stall whenever there is a strong imbalance between read and write accesses to the channel. Imbalances might be caused by channel reads or writes operating at different rates.

For example, if you find that the stall percentage of a write channel call is high, check to see if the occupancy and activity of the read channel call are low. If they are, the performing speed of the kernel controlling the read channel call is too slow for the kernel controlling the write channel call, leading to a performance bottleneck.

Related Information

- Transfer Data Via AOCL Channels or OpenCL Pipes on page 1-7
- Source Code Tab on page 1-22

Low Bandwidth Efficiency

Low bandwidth efficiency occurs when excessive amount of bandwidth is necessary to obtain useful data. Excessive bandwidth usage generally occurs when memory accesses are poor (for example, random accesses), leading to unsatisfactory coalescing opportunities.

Review your memory accesses to see if you can rewrite them such that accesses to memory sites address consecutive memory regions.

Related Information

- Strategies for Improving Memory Access Efficiency on page 1-62
- Source Code Tab on page 1-22

Stalling Channels

Channels provide a point-to-point communication link between either two kernels, or between a kernel and an I/O channel. If an I/O channel stalls, it implies that the I/O channel cannot keep up with the kernel.

For example, if a kernel has a read channel call to an Ethernet I/O and the profiler identifies a stall, it implies that the write channel is not writing data to the Ethernet I/O at the same rate as the read rate of the kernel.

For kernel-to-kernel channels, stalls occur if there is an imbalance between the read and write sides of the channel, or if the read and write kernels are not running concurrently.

For example, if the kernel that reads is not launched concurrently with the kernel that writes, or if the read operations occur much slower than the write operations, the Profiler identifies a stall for the `write_channel_altera` call in the write kernel.

Related Information

Transfer Data Via AOCL Channels or OpenCL Pipes on page 1-7

AOCL Profiler Limitations

The Altera SDK for OpenCL (AOCL) Profiler has some limitations.

- The Profiler can only extract one set of profile data from a kernel while it is running.

  If the Profiler collects the profile data after kernel execution completes, you can call the host application programming interface (API) to generate the **profile.mon** file multiple times.

  For more information on how to collect profile data during kernel execution, refer to the *Collecting Profile Data During Kernel Execution* section of the *Altera SDK for OpenCL Programming Guide*.
- Profile data is not persistent across OpenCL programs or multiple devices.

  You can request profile data from a single OpenCL program and on a single device only. If your host swaps a new kernel program in and out of the FPGA, the Profiler will not save the profile data.
- Instrumenting the Verilog code with performance counters increases hardware resource utilization (that is, FPGA area usage) and typically decreases performance.

  For information on instrumenting the Verilog code with performance counters, refer to the *Instrumenting the Kernel Pipeline with Performance Counters* section of the *Altera SDK for OpenCL Programming Guide*.

Related Information

- **Collecting Profile Data During Kernel Execution**
- **Instrumenting the Kernel Pipeline with Performance Counters (--profile)**

Strategies for Improving Single Work-Item Kernel Performance

Optimization Report on page 1-30
When you compile your OpenCL single work-item kernel, the Altera Offline Compiler (AOC) generates an optimization report that lists the problem areas, such as loop-carried dependencies, that might degrade performance.

Addressing Single Work-Item Kernel Dependencies Based on Optimization Report Feedback on page 1-38
In many cases, designing your OpenCL application as a single work-item kernel is sufficient to maximize performance without performing additional optimization steps.

Good Design Practices for Single Work-Item Kernel on page 1-50
If your OpenCL kernels contain loop structures, follow the Altera-recommended guidelines to construct the kernels in a way that allows the Altera Offline Compiler (AOC) to analyze them effectively.

Optimization Report

When you compile your OpenCL single work-item kernel, the Altera Offline Compiler (AOC) generates an optimization report that lists the problem areas, such as loop-carried dependencies, that might degrade performance.

By default, the optimization report does not contain source mapping information such as variable names and line numbers. To include the source mapping information in the optimization report, compile your kernel with the `-g` option of the `aoc` command. If you do not include the `-g` option, the report displays the following warning message:

```
==================================================================================
|                              *** Optimization Report ***                        |
| Warning: Compile with "-g" to get line number and variable name information     |
==================================================================================
| Kernel: myKernel                                                      | File:Ln  |
==================================================================================
```

Caution: Generating source mapping information increases the number of instructions. As a result, compilation time and memory usage might increase.

The information in the optimization report correlates to your kernel source code via a file index number and a line number. The file name that corresponds to each file index number appears at the bottom of the optimization report. For a design with multiple files, the file index number identifies the kernel and the line number locates the source code. Below is an example optimization report that includes source mapping information:

```
==================================================================================
|                              *** Optimization Report ***                        |
==================================================================================
| Kernel: test1                                                          | File:Ln |
==================================================================================
| Loop for.body.i                                                        | [3]:5   |
|   Pipelined execution inferred.                                        |         |
----------------------------------------------------------------------------------
| Loop for.body.i8                                                       | [2]:5   |
|   Pipelined execution inferred.                                        |         |
==================================================================================
| Kernel: test2                                                          | File:Ln |
==================================================================================
| Loop for.body.i                                                        | [2]:5   |
|   Pipelined execution inferred.                                        |         |
----------------------------------------------------------------------------------
| Loop for.body.i8                                                       | [3]:5   |
|   Pipelined execution inferred.                                        |         |
==================================================================================
| File Index:                                                                     |
==================================================================================
| [1]: <path_to_kernel>/main.cl                                                   |
| [2]: <path_to_kernel>/mul.cl                                                    |
| [3]: <path_to_kernel>/add.cl                                                    |
==================================================================================
```

The optimization report provides the following information for each loop in a single work-item kernel:

- Whether the AOC infers pipelined execution successfully.
- For each loop, the report identifies:
  - Operations that contribute the largest delay to the computation of the loop-carried dependency.
  - The launch frequency of a new loop iteration.
    - If a loop iteration launches every clock cycle, the kernel achieves maximum pipeline efficiency and yields the best performance.
    - If a loop iteration launches once every few clock cycles, it might degrade performance. The optimization report identifies the corresponding loop-carried dependency.

The optimization report addresses two types of loop-carried dependencies:

- Data dependencies

  The report identifies the variable name whose computation is dependent on other operations.

- Memory dependencies

  The report shows a memory dependency as a memory operation that is dependent on other operation(s).

Important: There might be other dependencies that do not affect the launch frequency of a new loop iteration but cause serial execution of certain regions of the kernel program. These dependencies also limit performance.

For more information on the `-g` option of the `aoc` command, refer to the *Adding Source References to Optimization Reports (-g)* section of the *Altera SDK for OpenCL Programming Guide*.

Related Information

Adding Source References to Optimization Reports (-g)

Optimization Report Messages

The single work-item kernel optimization report provides detailed information on the effectiveness of pipelined execution. It also identifies the nature of loop-carried dependencies to help you pinpoint the sources of performance bottleneck.

Optimization Report Messages for Simplified Analysis of a Complex Design on page 1-37
For complex designs that have long compilation times, the Altera Offline Compiler (AOC) performs simplified analyses for determining initiation interval (II) bottlenecks.

Optimization Report Messages for Pipelined Execution Inference

When the Altera Offline Compiler (AOC) attempts to infer pipelined execution, it generates a message in the optimization report in the ***<your_kernel_filename>/<your_kernel_filename>*.log** file.

Successful Pipelined Execution Inference

If the AOC successfully extracts the loop for pipelined execution, it prints the following message in the optimization report:

```
Pipelined execution inferred.
```

In the case where the AOC fails to infer pipelined execution, it specifies the reason of the failure in the optimization report message.

Unable to Resolve Loop Exit Condition at Iteration Initiation

The AOC normally evaluates the exit condition of loops at the end of the loop. During loop pipelining, the AOC moves these operations to the beginning of the loop where iteration initiation occurs. If the exit condition involves operations such as memory accesses, the AOC cannot move these operations safely.

Consider the following code example:

```
#define N 128
__kernel void exitcond( __global unsigned* restrict input,
                        __global unsigned* restrict result )
{
    unsigned i = 0;
    unsigned sum = 0;

    while( input[ i++ ] < N )
    {
        for ( unsigned j = 0; j < N; j++ )
        sum += input[i+j];
    }

    *result = sum;
}
```

The exit condition of the outer loop (that is, `input [ i++ ] < N`) contains a memory instruction. As such, the AOC cannot move the exit condition to the beginning of the outer loop. Therefore, the AOC cannot infer pipelining, and it prints the following message in the optimization report:

```
==================================================================================
| Kernel: exitcond                                                     | File:Ln  |
==================================================================================
| Loop for.cond.preheader                                              | [1]:9    |
|   Pipelined execution NOT inferred due to:                           |          |
|     Loop exit condition unresolvable at iteration initiation         |          |
----------------------------------------------------------------------------------
| Loop for.body                                                        | [1]:10   |
|   Pipelined execution inferred.                                      |          |
==================================================================================
```

Loop Structure Does Not Support Linear Execution

In order for the AOC to infer pipelined execution, loops must execute in a linear fashion. Consider the following code example:

```
__kernel void structure( __global unsigned* restrict output1,
                         __global unsigned* restrict output2 )
{
    for ( unsigned i = 0; i < N; i++ )
    {
        if ( (i & 3) == 0 )
        {
            for ( unsigned j = 0; j < N; j++ )
            {
                output1[ i+j ] = i * j;
            }
        }
        else
        {
            for ( unsigned j = 0; j < N; j++ )
            {
                output2[ i+j ] = i * j;
            }
        }
    }
}
```

The outer loop (`i`) contains two divergent inner loops. Each iteration of the outer loop may execute one inner loop or the other, which is a nonlinear execution. Therefore, the AOC cannot infer pipelining and prints the following message in the optimization report:

```
=================================================================================
| Kernel: structure                                                 | File:Ln   |
=================================================================================
| Loop for.body                                                     | [1]:20    |
|   Pipelined execution NOT inferred due to:                        |           |
|     Loop structure                                                |           |
---------------------------------------------------------------------------------
| Loop for.body8                                                    | [1]:27    |
|   Pipelined execution inferred.                                   |           |
---------------------------------------------------------------------------------
| Loop for.body4                                                    | [1]:22    |
|   Pipelined execution inferred.                                   |           |
=================================================================================
```

Out-of-Order Loop Iterations

A loop-carried dependency occurs when the iterations of an outer loop become out of order with respect to the inner loop. In such cases, the AOC cannot infer pipelining because it might lead to functionally incorrect results. Consider the following code example:

```
__kernel void order( __global unsigned* restrict input,
                     __global unsigned* restrict output )
{
    unsigned sum = 0;
    for ( unsigned i = 0; i < N; i++ )
    {
        for ( unsigned j = 0; j < i; j++ )
        {
            sum += input[ i+j ];
        }
    }
    output[ 0 ] = sum;
}
```

The number of iterations of the inner loop is different for each iteration of the outer loop. For example, for `i = 0`, `j` iterates zero times; for `i = i`, `j` iterates once, and so on. As such, the AOC cannot infer pipelining, and it prints the following message in the optimization report:

```
===================================================================================
| Kernel: order                                                       | File:Ln   |
===================================================================================
| Loop for.cond1.preheader                                            | [1]:38    |
|   Pipelined execution NOT inferred due to:                          |           |
|     Loop iteration ordering, iterations may get out of order        |           |
|     with respect to:                                                |           |
|       Loop for.body3                                                | [1]:39    |
-----------------------------------------------------------------------------------
| Loop for.body3                                                      | [1]:39    |
|   Pipelined execution inferred.                                     |           |
===================================================================================
```

Optimization Report Message Detailing Initiation Interval

The launch frequency of a new loop iteration is called the *initiation interval* (II).

The following optimization report message provides details of the II of a loop iteration:

`Successive iterations launched every <N> cycles due to:`

where <*N*> is the number of hardware clock cycles for which the pipeline must wait before it can process the next loop iteration. The ideal value for <*N*> is 1, which means II equals to 1.

Note: When II equals 1, the AOC does not display this optimization report message.

Optimization Report Messages for Loop-Carried Dependencies Affecting Initiation Interval

The optimization report messages provide details of data and memory dependencies that affect the initiation interval (II) of a pipelined loop.

Data Dependency

```
===================================================================================
|                       *** Optimization Report ***                               |
===================================================================================
| Kernel: myKernel                                                     | File:Ln |
===================================================================================
| Loop for.body                                                        | [1]:5   |
|     Pipelined execution inferred                                     |         |
|     Successive iterations launched every 9 cycles due to:            |         |
|                                                                      |         |
|         Data dependency on variable sum                              | [1]:4   |
|         Largest Critical Path Contributor:                           |         |
|            96%: Fadd Operation                                       | [1]:6   |
===================================================================================
| File Index:                                                                     |
===================================================================================
| [1]: <path_to_kernel>/<kernel_filename>.cl                                      |
===================================================================================
```

The message `Data dependency on variable <variable_name>` identifies a loop-carried dependency where the computation of a variable is dependent on the result from a previous loop iteration.

Memory Dependency

```
===================================================================================
|                        *** Optimization Report ***                              |
===================================================================================
| Kernel: myKernel                                                     | File:Ln |
===================================================================================
| Loop for.body                                                        | [1]:18  |
|     Pipelined execution inferred.                                    |         |
|     Successive iterations launched every 7 cycles due to:            |         |
|                                                                      |         |
|         Memory dependency on Load Operation from:                    | [1]:20  |
|           Store Operation                                            | [1]:20  |
|         Largest Critical Path Contributors:                          |         |
|           73%: Load Operation                                        | [1]:20  |
|           26%: Store Operation                                       | [1]:20  |
===================================================================================
| File Index:                                                                     |
===================================================================================
| [1]: <path_to_kernel>/<kernel_filename>.cl                                      |
===================================================================================
```

The message `Memory dependency on <memory_operation> from: <other_memory_operation>` identifies a loop-carried dependency where <*memory_operation*> cannot occur before the execution of <*other_memory_operation*> from a previous loop iteration completes.

The message `Largest Critical Path Contributor(s):` specifies the operations that contribute the largest delay to the computation of the loop-carried dependency.

Optimization Report Messages for Loop-Carried Dependencies Not Affecting the Initiation Interval

The optimization report generates messages that identify loop-carried dependencies not affecting the initiation interval (II) of a pipelined loop.

Consider the following optimization report example:

```
===================================================================================
|                        *** Optimization Report ***                              |
===================================================================================
| Kernel: myKernel                                                     | File:Ln |
===================================================================================
| Loop for.cond1.preheader                                             | [1]:6   |
|     Pipelined execution inferred.                                    |         |
|                                                                      |         |
|     Iterations will be executed serially across the following region: |        |
|                                                                      |         |
|         Loop for.body3                                               | [1]:7   |
|         due to:                                                      |         |
|         Data dependency on variable sum                              | [1]:5   |
===================================================================================
| File Index:                                                                     |
===================================================================================
| [1]: <path_to_kernel>/<kernel_filename>.cl                                      |
===================================================================================
```

The following message indicates that the computation of the variable or memory operation spans across the specified loop(s), resulting in the serial execution of the loop iterations over the specified region.

```
Iterations will be executed serially across the following region:
    Loop <loop_name>
    <Loop <loop_name_2>>
```

```
due to:
<data/memory_dependency_message>
```

where *<Loop <loop_name_2>>* is the name of the inner loop, if applicable, and *<data/memory_dependency_message>* is the data or memory dependency report message described in the *Optimization Report Messages for Loop-Carried Dependencies Affecting Initiation Interval* section.

There might be memory dependencies in the pipelined loop that might become performance bottlenecks after you address other performance bottlenecks in your optimizations. In this case, the optimization report might issue the following message:

```
===================================================================================
|                       *** Optimization Report ***                               |
===================================================================================
| Kernel: myKernel                                                    | File:Ln |
===================================================================================
| Loop for.body                                                       | [1]:14  |
|                                                                     |         |
|    Additional memory dependency:                                    |         |
|                                                                     |         |
|        Memory dependency on Load Operation from:                    | [1]:17  |
|          Store Operation                                            | [1]:17  |
===================================================================================
| File Index:                                                                     |
===================================================================================
| [1]: <path_to_kernel>/<kernel_filename>.cl                                      |
===================================================================================
```

Related Information

Optimization Report Messages for Loop-Carried Dependencies Affecting Initiation Interval on page 1-35

Optimization Report Messages for Simplified Analysis of a Complex Design

For complex designs that have long compilation times, the Altera Offline Compiler (AOC) performs simplified analyses for determining initiation interval (II) bottlenecks.

If the AOC has to analyze a simpler design model to identify II bottlenecks, it prints the following messages in the optimization report:

```
===================================================================================
|                           *** Optimization Report ***                           |
===================================================================================
| Kernel: test                                                      | File:Ln   |
===================================================================================
| Loop for.body                                                     | [1]:5     |
|   Pipelined execution inferred.                                   |           |
|   Successive iterations launched every 300 cycles due to:         |           |
|                                                                   |           |
|       Memory dependency on Load Operation from:                   | [1]:6     |
|         Store Operation                                           | [1]:6     |
|       Largest Critical Path Contributors:                         |           |
|           53%: Load Operation                                     | [1]:6     |
|           45%: Store Operation                                    | [1]:6     |
|       (Simple model used for II bottleneck estimation due to design |         |
|       complexity.)                                                |           |
===================================================================================
```

If the AOC cannot find the II bottleneck, it prints the following messages in the optimization report:

```
===================================================================================
|                            *** Optimization Report ***                          |
===================================================================================
| Kernel: test                                                        | File:Ln  |
===================================================================================
| Loop for.body                                                       | [1]:5    |
|   Pipelined execution inferred.                                     |          |
|   Successive iterations launched every 300 cycles due to:           |          |
|                                                                     |          |
|       Unable to determine exact cause                               |          |
===================================================================================
```

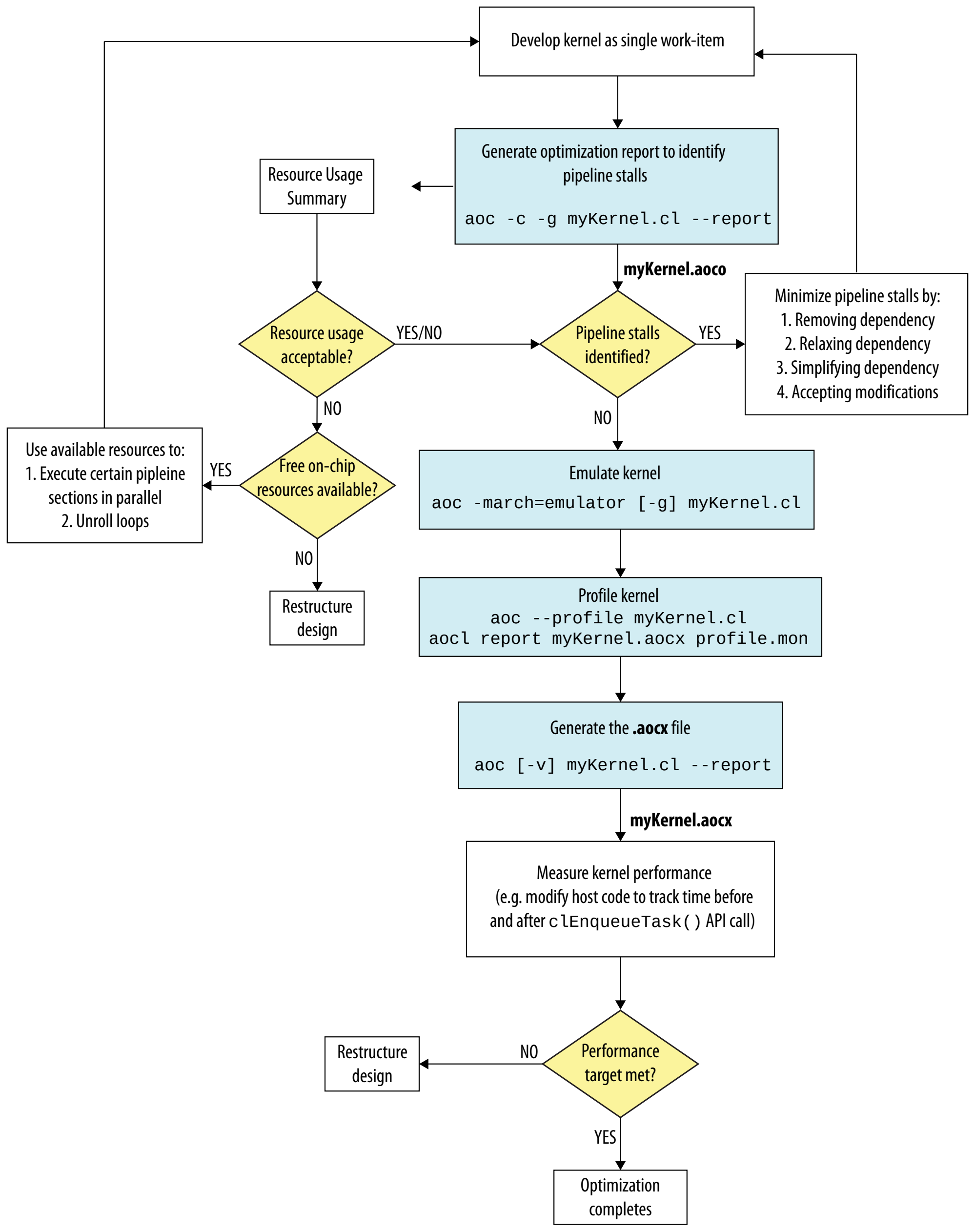
Addressing Single Work-Item Kernel Dependencies Based on Optimization Report Feedback

In many cases, designing your OpenCL application as a single work-item kernel is sufficient to maximize performance without performing additional optimization steps. To further improve the performance of your single work-item kernel, you can optimize it by addressing dependencies that the optimization report identifies.

The following flowchart outlines the approach you can take to iterate on your design and optimize your single work-item kernel. For usage information on the Altera SDK for OpenCL Emulator and the Profiler, refer to the *Emulating and Debugging Your OpenCL Kernel* and *Profiling Your OpenCL Kernel* sections of the *Altera SDK for OpenCL Programming Guide*, respectively. For information on the Profiler GUI and profiling information, refer to the *Profile Your Kernel to Identify Performance Bottlenecks* section.

Altera recommends the following optimization options to address single work-item kernel loop-carried dependencies, in order of applicability: removal, relaxation, simplification, and transfer to local memory.

Figure 1-14: Single Work-Item Optimization Work Flow


Develop kernel as single work-item
Generate optimization report to identify pipeline stalls
aoc -c -g myKernel.cl --report
Resource Usage Summary
myKernel.aoco
Resource usage acceptable?
YES/NO
Pipeline stalls identified?
YES
Minimize pipeline stalls by:
1. Removing dependency
2. Relaxing dependency
3. Simplifying dependency
4. Accepting modifications
NO
NO
Use available resources to:
1. Execute certain pipleine sections in parallel
2. Unroll loops
YES
Free on-chip resources available?
Emulate kernel
aoc -march=emulator [-g] myKernel.cl
NO
Restructure design
Profile kernel
aoc --profile myKernel.cl
aocl report myKernel.aocx profile.mon
Generate the .aocx file
aoc [-v] myKernel.cl --report
myKernel.aocx
Measure kernel performance
(e.g. modify host code to track time before and after clEnqueueTask() API call)
Restructure design
NO
Performance target met?
YES
Optimization completes

1. **Removing Loop-Carried Dependency** on page 1-40
   Based on the feedback from the optimization report, you can remove a loop-carried dependency by implementing a simpler memory access pattern.
2. **Relaxing Loop-Carried Dependency** on page 1-42
   Based on the feedback from the optimization report, you can relax a loop-carried dependency by increasing the dependence distance.
3. **Simplifying Loop-Carried Dependency** on page 1-44
   In cases where you cannot remove or relax the loop-carried dependency in your kernel, you might be able to simplify the dependency to improve single work-item kernel performance.

4. **Transferring Loop-Carried Dependency to Local Memory** on page 1-46
   For a loop-carried dependency that you cannot remove, improve the initiation interval (II) by moving the array with the loop-carried dependency from global memory to local memory.
5. **Removing Loop-Carried Dependency by Inferring Shift Registers** on page 1-48
   To enable the Altera Offline Compiler (AOC) to handle single work-item kernels that carry out double precision floating-point operations efficiently, remove loop-carried dependencies by inferring a shift register.

Related Information

- **Emulating and Debugging Your OpenCL Kernel**
- **Profiling Your OpenCL Kernel**
- **Profile Your Kernel to Identify Performance Bottlenecks** on page 1-21

Removing Loop-Carried Dependency

Based on the feedback from the optimization report, you can remove a loop-carried dependency by implementing a simpler memory access pattern.

Consider the following kernel:

```
#define N 128

__kernel void unoptimized (__global int * restrict A,
                           __global int * restrict B,
                           __global int* restrict result)
{
  int sum = 0;

  for (unsigned i = 0; i < N; i++) {
    for (unsigned j = 0; j < N; j++) {
      sum += A[i*N+j];
    }
    sum += B[i];
  }

  * result = sum;
}
```

The optimization report for kernel `unoptimized` resembles the following:

```
==================================================================================
|                        *** Optimization Report ***                             |
==================================================================================
| Kernel: unoptimized                                                | File:Ln |
==================================================================================
| Loop for.cond1.preheader                                           | [1]:9   |
|     Pipelined execution inferred.                                  |         |
|     Successive iterations launched every 2 cycles due to:          |         |
|                                                                    |         |
|         Pipeline structure                                         |         |
|                                                                    |         |
|     Iterations will be executed serially across the following region: |      |
|                                                                    |         |
|         Loop for.body3                                             | [1]:10  |
|         due to:                                                    |         |
|         Data dependency on variable sum                            | [1]:7   |
----------------------------------------------------------------------------------
| Loop for.body3                                                     | [1]:10  |
|     Pipelined execution inferred.                                  |         |
==================================================================================
| File Index:                                                                  |
================================================================================
| [1]: <path_to_kernel>/unoptimized.cl                                         |
================================================================================
```

- The first row of the report indicates that the Altera Offline Compiler (AOC) successfully infers pipelined execution for the outer loop, and a new loop iteration will launch every other cycle.
- The message `due to Pipeline structure` indicates that the AOC creates a pipeline structure that causes an outer loop iteration to launch every two cycles. The behavior is not a result of how you structure your kernel code.

  Note: For recommendations on how to structure your single work-item kernel, refer to the *Good Design Practices for Single Work-Item Kernel* section.
- The remaining messages in the first row of report indicate that the loop executes a single iteration at a time across the subloop because of data dependency on the variable *sum*. This data dependency exists because each outer loop iteration requires the value of *sum* from the previous iteration to return before the inner loop can start executing.
- The second row of the report notifies you that the inner loop executes in a pipelined fashion with no performance-limiting loop-carried dependencies.

To optimize the performance of this kernel, remove the data dependency on variable *sum* so that the outer loop iterations do not execute serially across the subloop. Perform the following tasks to decouple the computations involving *sum* in the two loops:

1. Define a local variable (for example, *sum2*) for use in the inner loop only.
2. Use the local variable from Step 1 to store the cumulative values of `A[i*N + j]` as the inner loop iterates.
3. In the outer loop, store the variable *sum* to store the cumulative values of `B[i]` and the value stored in the local variable.

Below is the restructured kernel `optimized`:

```
#define N 128

__kernel void optimized (__global int * restrict A,
                         __global int * restrict B,
                         __global int * restrict result)
{
  int sum = 0;

  for (unsigned i = 0; i < N; i++) {
    // Step 1: Definition
    int sum2 = 0;

    // Step 2: Accumulation of array A values for one outer loop iteration
    for (unsigned j = 0; j < N; j++) {
      sum2 += A[i*N+j];
    }

    // Step 3: Addition of array B value for an outer loop iteration
    sum += sum2;
    sum += B[i];
  }

  * result = sum;
}
```

An optimization report similar to the one below indicates the successful removal of the loop-carried dependency on the variable `sum`:

```
==================================================================================
|                        *** Optimization Report ***                             |
==================================================================================
| Kernel: optimized                                                    | File:Ln |
==================================================================================
| Loop for.cond1.preheader                                             | [1]:9   |
|     Pipelined execution inferred.                                    |         |
|     Successive iterations launched every 2 cycles due to:            |         |
|                                                                      |         |
|         Pipeline structure                                           |         |
----------------------------------------------------------------------------------
| Loop for.body3                                                       | [1]:14  |
|     Pipelined execution inferred.                                    |         |
==================================================================================
| File Index:                                                                    |
==================================================================================
| [1]: <path_to_kernel>/optimized.cl                                             |
==================================================================================
```

You have addressed all the loop-carried dependence issues successfully when you see only the following messages in the optimization report:

- `Pipelined execution inferred` for innermost loops.
- `Pipelined execution inferred. Successive iterations launched every 2 cycles due to: Pipeline structure` for all other loops.

Related Information

Good Design Practices for Single Work-Item Kernel on page 1-50

Relaxing Loop-Carried Dependency

Based on the feedback from the optimization report, you can relax a loop-carried dependency by increasing the dependence distance.Increase the dependence distance by increasing the number of loop iterations that occurs between the generation of a loop-carried value and its usage.

Consider the following code example:

```
 1 #define N 128
 2
 3 __kernel void unoptimized (__global float * restrict A,
 4                            __global float * restrict result)
 5 {
 6   float mul = 1.0f;
 7
 8   for (unsigned i = 0; i < N; i++)
 9     mul *= A[i];
10
11   * result = mul;
12 }
```

```
==================================================================================
|                        *** Optimization Report ***                             |
==================================================================================
| Kernel: unoptimized                                                  | File:Ln |
==================================================================================
| Loop for.body                                                        | [1]:8   |
|     Pipelined execution inferred.                                    |         |
|     Successive iterations launched every 3 cycles due to:            |         |
|                                                                          |         |
|         Data dependency on variable mul                                  | [1]:9   |
|         Largest Critical Path Contributor:                               |         |
|             100%: Fmul Operation                                         | [1]:9   |
===================================================================================
| File Index:                                                                      |
===================================================================================
| [1]: <path_to_kernel>/unoptimized.cl                                             |
===================================================================================
```

The optimization report above shows that the Altera Offline Compiler (AOC) infers pipelined execution for the loop successfully. However, the loop-carried dependency on the variable *mul* causes loop iterations to launch every three cycles. In this case, the floating-point multiplication operation on line 9 (that is, `mul *= A[i]`) contributes the largest delay to the computation of the variable *mul*.

To relax the loop-carried data dependency, instead of using a single variable to store the multiplication results, operate on *M* copies of the variable and use one copy every *M* iterations:

1. Declare multiple copies of the variable *mul* (for example, in an array called *mul_copies*).
2. Initialize all the copies of *mul_copies*.
3. Use the last copy in the array in the multiplication operation.
4. Perform a shift operation to pass the last value of the array back to the beginning of the shift register.
5. Reduce all the copies to *mul* and write the final value to *result*.

Below is the restructured kernel:

```
#define N 128
#define M 6

__kernel void optimized (__global float * restrict A,
                         __global float * restrict result)
{
  float mul = 1.0f;

  // Step 1: Declare multiple copies of variable mul
  float mul_copies[M];

  // Step 2: Initialize all copies
  for (unsigned i = 0; i < M; i++)
    mul_copies[i] = 1.0f;

  for (unsigned i = 0; i < N; i++) {
    // Step 3: Perform multiplication on the last copy
    float cur = mul_copies[M-1] * A[i];

    // Step 4a: Shift copies
    #pragma unroll 5
    for (unsigned j = M-1; j > 0; j--)
      mul_copies[j] = mul_copies[j-1];

    // Step 4b: Insert updated copy at the beginning
    mul_copies[0] = cur;
  }

  // Step 5: Perform reduction on copies
  #pragma unroll 6
  for (unsigned i = 0; i < M; i++)
    mul *= mul_copies[i];

  * result = mul;
}
```

An optimization report similar to the one below indicates the successful relaxation of the loop-carried dependency on the variable *mul*:

```
==================================================================================
|                       *** Optimization Report ***                              |
==================================================================================
| Kernel: optimized                                                    | File:Ln |
==================================================================================
| Loop for.body4                                                       | [1]:16  |
|     Pipelined execution inferred.                                    |         |
==================================================================================
| File Index:                                                                    |
==================================================================================
| [1]: <path_to_kernel>/optimized.cl                                             |
==================================================================================
```

Simplifying Loop-Carried Dependency

In cases where you cannot remove or relax the loop-carried dependency in your kernel, you might be able to simplify the dependency to improve single work-item kernel performance.

Consider the following kernel example:

```
#define N 128
#define NUM_CH 3

channel uchar CH_DATA_IN[NUM_CH];
channel uchar CH_DATA_OUT;

__kernel void unoptimized()
{
  unsigned storage = 0;
  unsigned num_bytes = 0;

  for (unsigned i = 0; i < N; i++) {

    #pragma unroll
    for (unsigned j = 0; j < NUM_CH; j++) {
      if (num_bytes < NUM_CH) {
        bool valid = false;
        uchar data_in = read_channel_nb_altera(CH_DATA_IN[j], &valid);
        if (valid) {
          storage <<= 8;
          storage |= data_in;
          num_bytes++;
        }
      }
    }

    if (num_bytes >= 1) {
      num_bytes -= 1;
      uchar data_out = storage >> (num_bytes*8);
      write_channel_altera(CH_DATA_OUT, data_out);
    }
  }
}
```

This kernel reads one byte of data from three input channels in a nonblocking fashion. It then writes the data one byte at a time to an output channel. It uses the variable *storage* to store up to 4 bytes of data, and uses the variable *num_bytes* to keep track of how many bytes are stored in *storage*. If *storage* has space available, then the kernel reads a byte of data from one of the channels and stores it in the least significant byte of *storage*.

The optimization report below indicates that there is a loop-carried dependency on the variable *num_bytes*:

```
===================================================================================
|                            *** Optimization Report ***                          |
===================================================================================
| Kernel: unoptimized                                                   | File:Ln |
===================================================================================
| Loop for.cond1.preheader                                              | [1]:12  |
|     Pipelined execution inferred.                                     |         |
|     Successive iterations launched every 7 cycles due to:             |         |
|                                                                       |         |
|         Data dependency on variable num_bytes                         | [1]:10  |
|         Largest Critical Path Contributors:                           |         |
|             16%: Icmp Operation                                       | [1]:16  |
|             16%: Icmp Operation                                       | [1]:16  |
|             16%: Icmp Operation                                       | [1]:16  |
|              8%: Icmp Operation                                       | [1]:27  |
|              7%: Add Operation                                        | [1]:22  |
|              7%: Add Operation                                        | [1]:22  |
|              7%: Add Operation                                        | [1]:22  |
|              3%: Non-Blocking Channel Read Operation                  | [1]:18  |
===================================================================================
| File Index:                                                                     |
===================================================================================
| [1]: <path_to_kernel>/unoptimized.cl                                            |
===================================================================================
```

The computation path of *num_bytes* is as follows:

1. Comparison on line 16 (`if (num_bytes < NUM_CH)`).
2. Computation of variable *valid* by the nonblocking channel read operation on line 18 (`uchar data_in = read_channel_nb_altera(CH_DATA_IN[j], &valid)`) for the comparison on line 19.
3. Addition on line 22 (`num_bytes++`).
4. Comparison on line 27 (`if (num_bytes >= 1)`).
5. Subtraction on line 28 (`num_bytes -= 1`).

Because of the `unroll` pragma on line 14, the Altera Offline Compiler (AOC) unrolls the loop, causing the comparisons and additions in the loop body to replicate three times. The optimization report shows that the comparisons are the most expensive operations on the computation path of *num_bytes*, followed by the additions on line 22.

To simplify the loop-carried dependency on *num_bytes*, consider restructuring the application to perform the following tasks:

1. Ensure that the kernel reads from the channels only if there is enough space available in *storage*, in the event that all channel read operations return data (that is, there is at least 3 bytes of empty space in *storage*).
   Setting this condition simplifies the computation path of the variable *num_bytes* by reducing the number of comparisons.
2. Increase the size of *storage* from 4 bytes to 8 bytes to satisfy the 3-byte space threshold more easily.

Below is the restructured kernel `optimized`:

```
1 #define N 128
2 #define NUM_CH 3
3
4 channel uchar CH_DATA_IN[NUM_CH];
5 channel uchar CH_DATA_OUT;
6
7 __kernel void optimized()
```

```
 8 {
 9   // Change storage to 64 bits
10   ulong storage = 0;
11   unsigned num_bytes = 0;
12
13   for (unsigned i = 0; i < N; i++) {
14
15     // Ensure that we have enough space if we read from ALL channels
16     if (num_bytes <= (8-NUM_CH)) {
17       #pragma unroll
18       for (unsigned j = 0; j < NUM_CH; j++) {
19         bool valid = false;
20         uchar data_in = read_channel_nb_altera(CH_DATA_IN[j], &valid);
21         if (valid) {
22           storage <<= 8;
23           storage |= data_in;
24           num_bytes++;
25         }
26       }
27     }
28
29     if (num_bytes >= 1) {
30       num_bytes -= 1;
31       uchar data_out = storage >> (num_bytes*8);
32       write_channel_altera(CH_DATA_OUT, data_out);
33     }
34   }
35 }
```

An optimization report similar to the one below indicates the successful simplification of the loop-carried dependency on the variable *num_bytes*:

```
==================================================================================
|                       *** Optimization Report ***                              |
==================================================================================
| Kernel: optimized                                                   | File:Ln |
==================================================================================
| Loop for.body                                                       | [1]:13  |
|     Pipelined execution inferred.                                   |         |
==================================================================================
| File Index:                                                                    |
==================================================================================
| [1]: <path_to_kernel>/optimized.cl                                             |
==================================================================================
```

Transferring Loop-Carried Dependency to Local Memory

For a loop-carried dependency that you cannot remove, improve the initiation interval (II) by moving the array with the loop-carried dependency from global memory to local memory.

Consider the following kernel example:

```
#define N 128

__kernel void unoptimized( __global int* restrict A,
                           __global int* restrict result )
{
    for (unsigned i = 1; i < N; i++)
        A[i] = A[i-1];

    *result = A[N-1];
}
```

```
==================================================================================
|                             *** Optimization Report ***                         |
==================================================================================
| Kernel: unoptimized                                                 | File:Ln |
==================================================================================
| Loop for.body                                                       | [1]:6   |
|   Pipelined execution inferred.                                     |         |
|   Successive iterations launched every 324 cycles due to:           |         |
|                                                                     |         |
|       Memory dependency on Load Operation from:                     | [1]:7   |
|         Store Operation                                             | [1]:7   |
|       Largest Critical Path Contributors:                           |         |
|           49%: Load Operation                                       | [1]:7   |
|           49%: Store Operation                                      | [1]:7   |
==================================================================================
| File Index:                                                                     |
==================================================================================
| [1]: <path_to_kernel>/unoptimized.cl                                            |
==================================================================================
```

Global memory accesses have long latencies. In this example, the loop-carried dependency on the array A[i] causes the long latency. This latency is reflected by an II of 324 in the optimization report. To reduce the II value by transferring the loop-carried dependency from global memory to local memory, perform the following tasks:

1. Copy the array with the loop-carried dependency to local memory. In this example, array A[i] becomes array B[i] in local memory.
2. Execute the loop with the loop-carried dependence on array B[i].
3. Copy the array back to global memory.

When you transfer array A[i] to local memory and it becomes array B[i], the loop-carried dependency is now on B[i]. Because local memory has a much lower latency than global memory, the II value improves.

Below is the restructured kernel optimized:

```
#define N 128

__kernel void optimized( __global int* restrict A,
                         __global int* restrict result )
{
    int B[N];

    for (unsigned i = 0; i < N; i++)
        B[i] = A[i];

    for (unsigned i = 1; i < N; i++)
        B[i] = B[i-1];

    for (unsigned i = 0; i < N; i++)
        A[i] = B[i];

    *result = B[N-1];
}
```

An optimization report similar to the one below indicates the successful reduction of II from 324 to 7:

```
==================================================================================
|                             *** Optimization Report ***                         |
==================================================================================
| Kernel: optimized                                                    | File:Ln |
==================================================================================
| Loop for.body                                                        | [1]:8   |
|   Pipelined execution inferred.                                      |         |
----------------------------------------------------------------------------------
| Loop for.body6                                                       | [1]:11  |
|   Pipelined execution inferred.                                      |         |
|   Successive iterations launched every 7 cycles due to:              |         |
|                                                                      |         |
|       Memory dependency on Load Operation from:                      | [1]:12  |
|         Store Operation                                              | [1]:12  |
|       Largest Critical Path Contributors:                            |         |
|           73%: Load Operation                                        | [1]:12  |
|           26%: Store Operation                                       | [1]:12  |
----------------------------------------------------------------------------------
| Loop for.body17                                                      | [1]:14  |
|   Pipelined execution inferred.                                      |         |
==================================================================================
| File Index:                                                                    |
==================================================================================
| [1]: <path_to_kernel>/optimized.cl                                             |
==================================================================================
```

Removing Loop-Carried Dependency by Inferring Shift Registers

To enable the Altera Offline Compiler (AOC) to handle single work-item kernels that carry out double precision floating-point operations efficiently, remove loop-carried dependencies by inferring a shift register.

Consider the following kernel:

```
 1 __kernel void double_add_1 (__global double *arr,
 2                             int N,
 3                             __global double *result)
 4 {
 5   double temp_sum = 0;
 6
 7   #pragma unroll
 8   for (int i = 0; i < N; ++i)
 9   {
10       temp_sum += arr[i];
11   }
12
13   *result = temp_sum;
14 }
```

The optimization report for kernel `unoptimized` resembles the following:

```
==================================================================================
|                        *** Optimization Report ***                             |
==================================================================================
| Kernel: unoptimized                                                  | File:Ln |
==================================================================================
| Loop for.body                                                        | [1]:8   |
|   Pipelined execution inferred.                                      |         |
|   Successive iterations launched every 12 cycles due to:             |         |
|                                                                      |         |
|       Data dependency on variable temp_sum                           | [1]:10  |
|       Largest Critical Path Contributor:                             |         |
|           97%: Fadd Operation                                        | [1]:10  |
==================================================================================
| File Index:                                                                    |
==================================================================================
| [1]: <path_to_kernel>/unoptimized.cl                                            |
===================================================================================
```

The kernel unoptimized is an accumulator that sums the elements of a double precision floating-point array `arr[i]`. For each loop iteration, the AOC takes 12 cycles to compute the result of the addition and then stores it in the variable *temp_sum*. Each loop iteration requires the value of *temp_sum* from the previous loop iteration, which creates a data dependency on *temp_sum*.

- To remove the data dependency, infer the array `arr[i]` as a shift register.

Below is the restructured kernel `optimized`:

```
//Shift register size must be statically determinable
#define II_CYCLES 12

__kernel void double_add_2 (__global double *arr,
                            int N,
                            __global double *result)
{
    //Create shift register with II_CYCLE+1 elements
    double shift_reg[II_CYCLES+1];

    //Initialize all elements of the register to 0
    for (int i = 0; i < II_CYCLES + 1; i++)
    {
        shift_reg[i] = 0;
    }

    //Iterate through every element of input array
    for(int i = 0; i < N; ++i)
    {
        //Load ith element into end of shift register
        //if N > II_CYCLE, add to shift_reg[0] to preserve values
        shift_reg[II_CYCLES] = shift_reg[0] + arr[i];

        #pragma unroll
        //Shift every element of shift register
        for(int j = 0; j < II_CYCLES; ++j)
        {
            shift_reg[j] = shift_reg[j + 1];
        }
    }

    //Sum every element of shift register
    double temp_sum = 0;

    #pragma unroll
    for(int i = 0; i < II_CYCLES; ++i)
    {
        temp_sum += shift_reg[i];
    }

    *result = temp_sum;
}
```

The following optimization report indicates that the inference of the shift register `shift_reg[II_CYCLES]` successfully removes the data dependency on the variable *temp_sum*:

```
===================================================================================
|                       *** Optimization Report ***                               |
===================================================================================
| Kernel: optimized                                                   | File:Ln |
===================================================================================
| Loop for.body4                                                      | [1]:18  |
|   Pipelined execution inferred.                                     |         |
==================================================================================
| File Index:                                                                    |
==================================================================================
| [1]: <path_to_kernel>/optimized.cl                                             |
==================================================================================
```

Good Design Practices for Single Work-Item Kernel

If your OpenCL kernels contain loop structures, follow the Altera-recommended guidelines to construct the kernels in a way that allows the Altera Offline Compiler (AOC) to analyze them effectively. Well-structured loops are particularly important when you direct the AOC to perform pipeline parallelism execution in loops.

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:

```
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:

```
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 that the AOC can achieve, which reduces kernel performance.

```
for(int i = 0; i < N; i++)
{
    A[i] = A[i - 1] + i;
}
```

The AOC performs a static memory dependence analysis on loops to determine the extent of parallelism that 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.*

  Compiler output is suboptimal when the kernel accesses arrays by dereferencing pointer values derived from arithmetic operations. For example, avoid accessing an array in the following manner:

  ```
  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:

Nested Loop	Converted Single Loop

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

Declare Variables in the Deepest Scope Possible

To reduce the hardware resources necessary for implementing a variable, declare the variable prior to its use in a loop. Declaring variables in the deepest scope possible minimizes data dependencies and hardware usage because the AOC does not need to preserve the variable data across loops that do not use the variables.

Consider the following example:

```
int a[N];
for (int i = 0; i < m; ++i)
{
    int b[N];
    for (int j = 0; j < n; ++j)
    {
        // statements
    }
}
```

The array a requires more resources to implement than the array b. To reduce hardware usage, declare array a outside the inner loop unless it is necessary to maintain the data through iterations of the outer loop.

Tip: Overwriting all values of a variable in the deepest scope possible also reduces the resources necessary to present the variable.

Strategies for Improving NDRange Kernel Data Processing Efficiency

Consider the following kernel code:

```
__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 1-53
   Specify the `max_work_group_size` or `reqd_work_group_size` attribute for your kernels whenever possible. These attributes allow the Altera Offline Compiler (AOC) to perform aggressive optimizations to match the kernel to hardware resources without any excess logic.
2. **Kernel Vectorization** on page 1-54
   Kernel vectorization allows multiple work-items to execute in a single instruction multiple data (SIMD) fashion.
3. **Multiple Compute Units** on page 1-57
   To achieve higher throughput, the Altera Offline Compiler (AOC) can generate multiple compute units for each kernel.
4. **Combination of Compute Unit Replication and Kernel SIMD Vectorization** on page 1-59
   If your replicated or vectorized OpenCL kernel does not fit in the FPGA, you can modify the kernel by both replicating the compute unit and vectorizing the kernel.
5. **Resource-Driven Optimization** on page 1-61
   The Altera Offline Compiler (AOC) analyzes automatically the effects of combining various values of kernel attributes and performs resource-driven optimizations.

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 Altera Offline Compiler (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 does not contain any 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` application programming interface (API) call to determine the work-group size constraints that 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.

Caution: 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`.

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 `clGetKernelWork-GroupInfo` 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.
- 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.

When queuing an NDRange kernel (that is, not a single work-item kernel), specify an explicit work-group size under the following conditions:

- If your kernel uses memory barriers, local memory, or local work-item IDs.
- If your host program queries the work-group size.

If your kernel uses memory barriers, perform one of the following tasks to minimize hardware resources:

- Specify a value for the `reqd_work_group_size` attribute.
- Assign to the `max_work_group_size` attribute the smallest work-group size that accommodates all your runtime work-group size requests.

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 improved efficiency in the implementation of kernel compute units. 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:

```
__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];
}
```

Kernel Vectorization

To achieve higher throughput, you can vectorize your kernel. Kernel vectorization allows multiple work-items to execute in a single instruction multiple data (SIMD) fashion. You can direct the Altera Offline Compiler (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:

```
__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];
}
```

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 work-group size you specify for reqd_work_group_size must be divisible by the value you assign to num_simd_work_items. 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:

```
__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 manually direct the AOC to coalesce each group of four load and store operations.

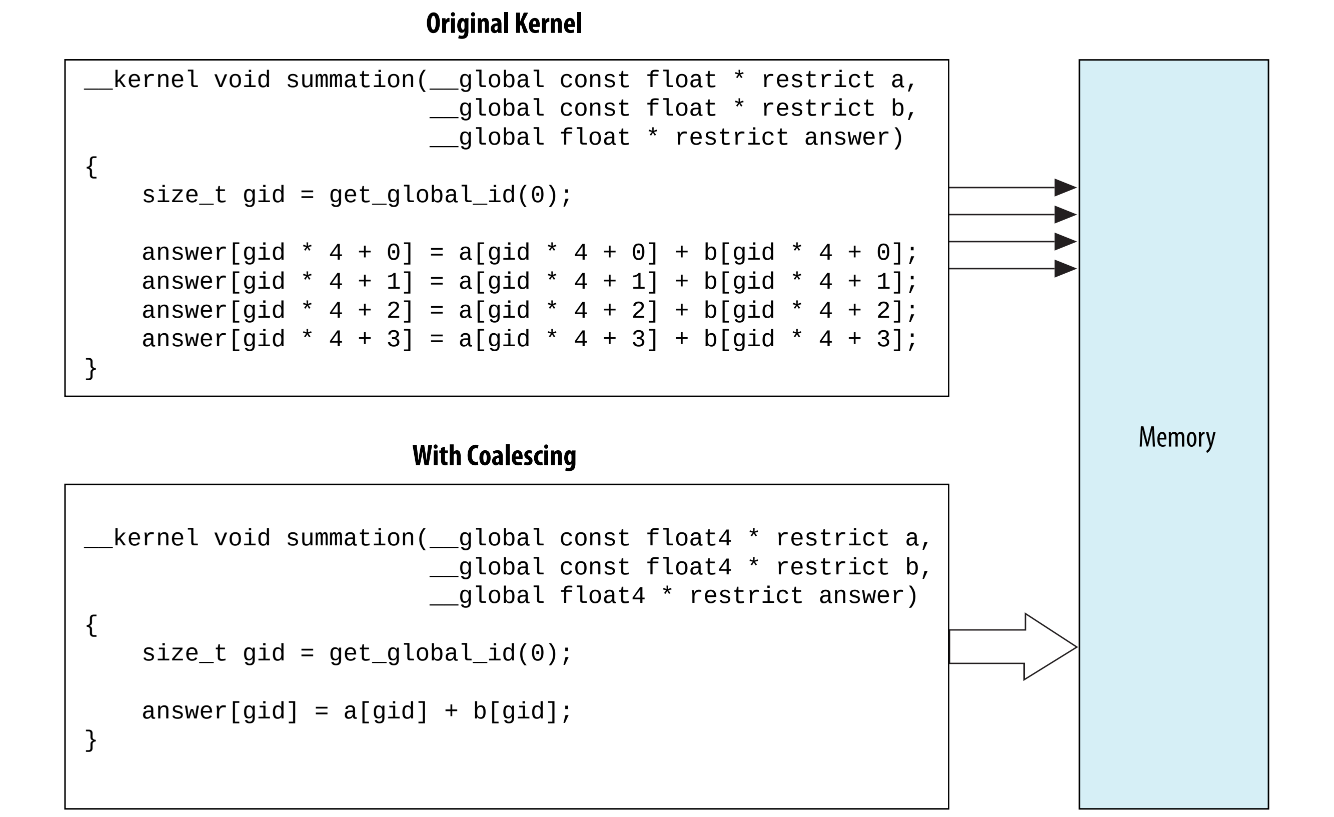
Attention: Each work-item handles four times as much work after you implement the manual optimizations. As a result, the host application must use an NDRange that is four times smaller than in the original example. On the contrary, you do not need to adjust the NDRange size when you exploit the automatic 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 Altera Offline Compiler (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 1-15: Static Memory Coalescing



```
__kernel void summation(__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];
}
```

```
__kernel void summation(__global const float4 * restrict a,
                        __global const float4 * restrict b,
                        __global float4 * restrict answer)
{
    size_t gid = get_global_id(0);

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

Consider the following vectorized kernel:

```
__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];
}
```

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];
}
```

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 Altera Offline Compiler (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.

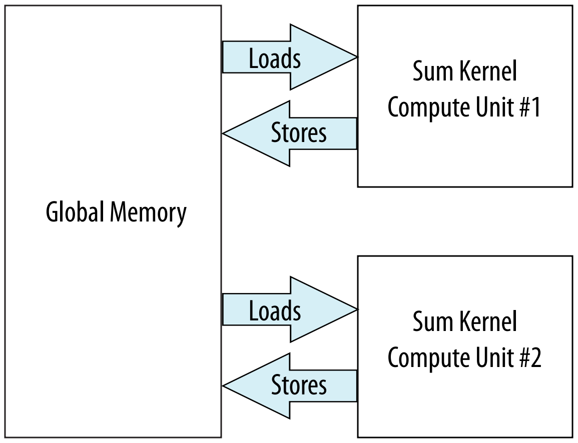
Assume each work-group takes the same amount of time to complete its execution. If the AOC implements two compute units, each compute unit executes half of the work-groups. Because the hardware scheduler dispatches the work-groups, you do not need to manage this process in your own code.

The AOC does not automatically determine 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 that the AOC should create using 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];
}
```

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

Figure 1-16: Data Flow with Multiple Compute Units



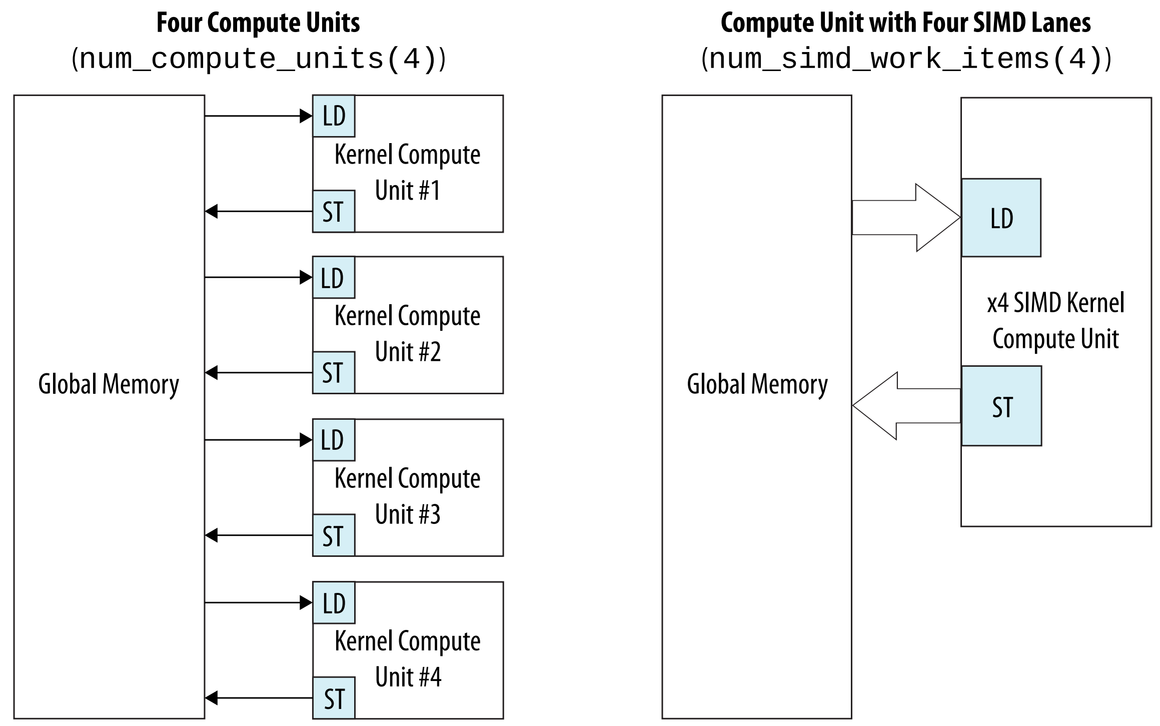
Compute Unit Replication versus Kernel SIMD Vectorization

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 that the Altera Offline Compiler (AOC) uses to implement your kernel. The `num_compute_units` attribute modifies the number of compute units to which work-groups can be scheduled, which also modifies the number of times a kernel accesses global memory. 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 single instruction multiple data (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.

Figure 1-17: Compute Unit Replication versus Kernel SIMD Vectorization



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

If your replicated or vectorized OpenCL kernel does not fit in the FPGA, you can modify the kernel by both replicating the compute unit and vectorizing the kernel. Include the `num_compute_units` attribute to modify the number of compute units for the kernel, and include the `num_simd_work_items` attribute to take advantage of kernel vectorization.

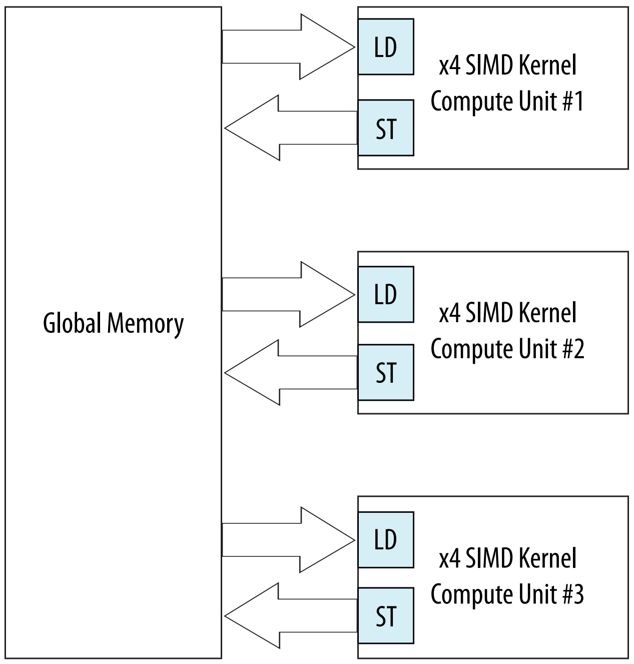
Consider a case where a kernel with a `num_simd_work_items` attribute set to 16 does not fit in the FPGA. The kernel might fit if you modify it by duplicating a narrower single instruction multiple data (SIMD) kernel compute unit. Determining the optimal balance between the number of compute units and 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:

```
__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 1-18: Optimizing Throughput by Combining Compute Unit Replication and Kernel SIMD Vectorization


Global Memory
LD
ST
x4 SIMD Kernel Compute Unit #1
LD
ST
x4 SIMD Kernel Compute Unit #2
LD
ST
x4 SIMD Kernel Compute Unit #3

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

The Altera Offline Compiler (AOC) analyzes automatically the effects of combining various values of kernel attributes and performs resource-driven optimizations.

During compilation, the AOC examines multiple values of the `num_compute_units` and `num_simd_work_items` kernel attributes in various combinations, and applies a set of heuristics to improve a base design incrementally. 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 following are examples of resource-driven optimization:

- 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, it applies optimizations that increase performance.

- In a multi-kernel design, improves the kernel(s) with minimum performance first.

  The order in which kernel optimization occurs is based on the work-items per second metric. When these kernels cannot be optimized any further, subsequent kernels are improved in order of their throughput estimates. During resource-driven optimization, the AOC maintains a set of high-performance candidates and attempts to apply incremental optimizations to each of them. Loop unrolling and single instruction multiple data (SIMD) vectorization are the preferred optimization strategies over compute unit replication because these optimizations generally result in more efficient hardware implementations.

- During resource-driven optimization, the AOC iterates on a predetermined set of optimization steps.

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

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 AOC assumes that loops with unknown bounds iterate 1024 times. Based on these assumptions, the AOC 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 do not want to unroll 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 accurately capture the maximum operating frequency that the hardware compiler achieves. Similarly, the estimated resource usage used in resource-driven optimization might not reflect the actual hardware resource usage.

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.

Strategies for Improving 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 Altera Offline Compiler (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 (Fmax), which degrades kernel perform- ance.

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 1-62
   Optimizing the memory accesses in your OpenCL kernels can improve overall kernel performance.
2. **Optimize Global Memory Accesses** on page 1-63
   The AOC interleaves global memory across each of the external memory banks.
3. **Perform Kernel Computations Using Constant, Local or Private Memory** on page 1-66
   To optimize memory access efficiency, minimize the number for global memory accesses by performing your OpenCL kernel computations in constant, local, or private memory.

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 logically separate 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 Altera Offline Compiler (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.

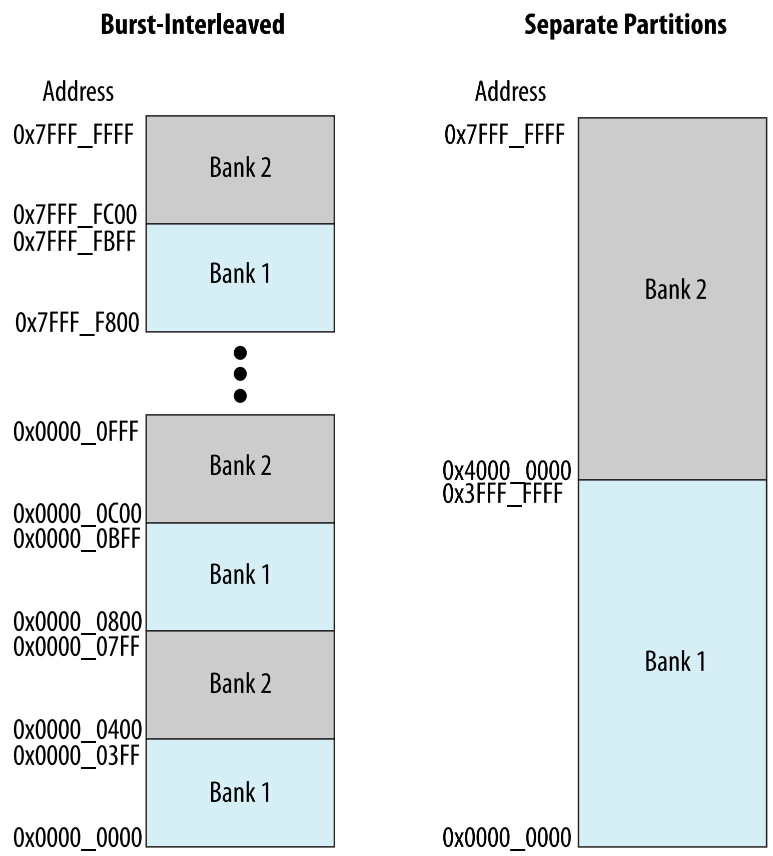
Optimize Global Memory Accesses

The Altera Offline Compiler (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.

Figure 1-19: Global 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 Altera Offline Compiler (AOC) directs the kernel to access consecutive locations in global memory.

Consider the following code example:

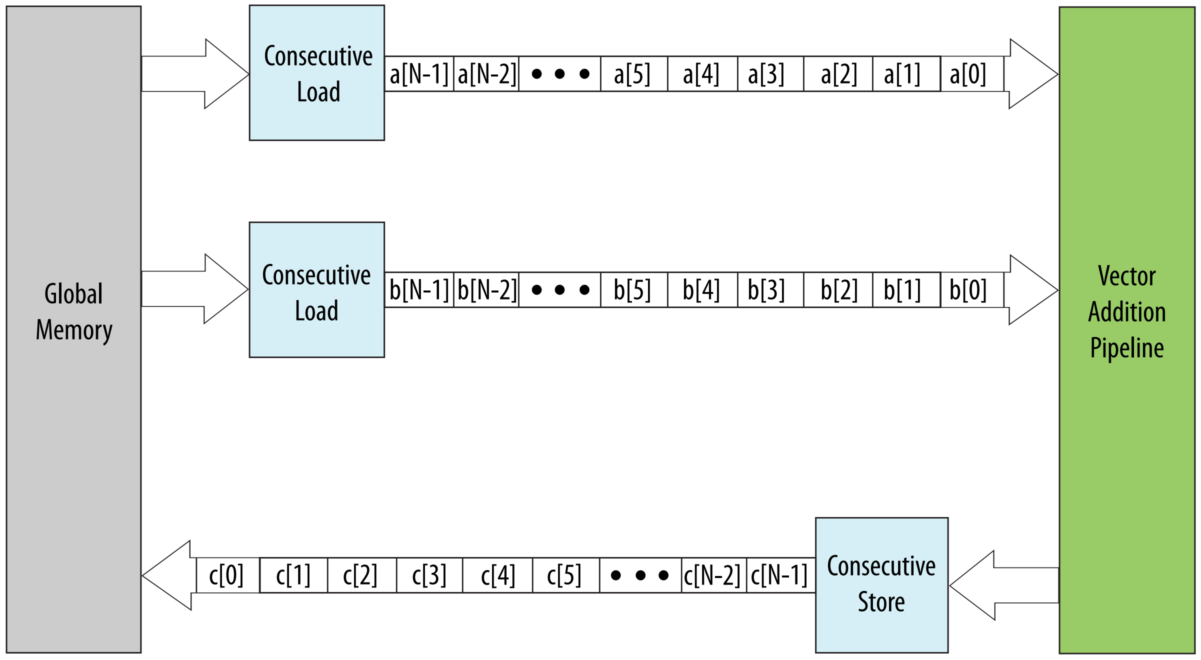
```
__kernel void sum ( __global const float * restrict a,
                    __global const float * restrict b,
                    __global float * restrict c )
{
    size_t gid = get_global_id(0);

    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 that exits 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 1-20: Contiguous Memory Access


Global Memory
Consecutive Load
a[N-1] a[N-2] • • • a[5] a[4] a[3] a[2] a[1] a[0]
Consecutive Load
b[N-1] b[N-2] • • • b[5] b[4] b[3] b[2] b[1] b[0]
Vector Addition Pipeline
c[0] c[1] c[2] c[3] c[4] c[5] • • • c[N-2] c[N-1]
Consecutive Store

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.

- The Altera Offline Compiler (AOC) cannot burst-interleave across different memory types. To manually partition a specific type of global memory , compile your OpenCL kernels with the `--no-interleaving <global_memory_type>` flag to configure each bank of a certain memory type as 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.

  If your kernel accesses heterogeneous global memory types, include the `--no-interleaving <global_memory_type>` option in the `aoc` command for each memory type that you want to partition manually.

For more information on the usage of the `--no-interleaving <global_memory_type>` option, refer to the *Disabling Burst-Interleaving of Global Memory (--no-interleaving <global_memory_type>)* section of the *Altera SDK for OpenCL Programming Guide.*

Related Information

Disabling Burst-Interleaving of Global Memory (--no-interleaving <global_memory_type>)

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.

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.

For more information on how to allocate buffers in global memory and how to modify your host application to leverage heterogeneous buffers, refer to the *Specifying Buffer Location in Global Memory* and *Allocating OpenCL Buffer for Manual Partitioning of Global Memory* sections of the *Altera SDK for OpenCL Programming Guide.*

Related Information

- **Specifying Buffer Location in Global Memory**
- **Allocating OpenCL Buffer for Manual Partitioning of Global Memory**

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>` option 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 that is already loaded into the constant cache, the cached data is discarded (that is, invalidated) from the constant cache.

For more information on the `--const-cache-bytes <N>` option, refer to the *Configuring Constant Memory Cache Size (--const-cache-bytes <N>)* section of the *Altera SDK for OpenCL Programming Guide*.

Related Information

Configuring Constant Memory Cache Size (--const-cache-bytes <N>)

Preloading Data to Local Memory

Local memory is considerably smaller than global memory, but it has significantly higher throughput and much lower latency. Unlike global memory accesses, the kernel can access local memory randomly without any performance penalty. When you structure 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 Altera Offline Compiler (AOC) implements OpenCL local memory in on-chip memory blocks in the FPGA. On-chip memory blocks have two read and write ports, and they 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 the closest value that is 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.

- When accessing local memory, use the simplest address calculations possible and avoid pointer math operations that are not mandatory.

  Altera recommends this coding style to reduce FPGA resource utilization and increase local memory efficiency by allowing the AOC to make better guarantees about access patterns through static code analysis. Complex address calculations and pointer math operations can prevent the AOC from creating independent memory systems representing different portions of your data, leading to increased area usage and decreased runtime performance.

- Avoid storing pointers to memory whenever possible. Stored pointers often prevent static compiler analysis from determining the data sets accessed, when the pointers are subsequently retrieved from memory. Storing pointers to memory almost always leads to suboptimal area and performance results

For usage information on the `local_mem_size` attribute, refer to the *Specifying Pointer Size in Local Memory* section of the *Altera SDK for OpenCL Programming Guide*.

Related Information

Specifying Pointer Size in Local Memory

Storing Variables and Arrays in Private Memory

The Altera Offline Compiler (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).

For more information on the implementation of private memory using registers, refer to the *Inferring a Register* section of the *Altera SDK for OpenCL Programming Guide*.

Related Information

Inferring a Register

Strategies for Optimizing FPGA Area Usage

Area usage is an important design consideration if your OpenCL kernels are executable on FPGAs of different sizes. When you design your OpenCL application, Altera recommends that you follow certain design strategies for optimizing hardware area usage.

Optimizing kernel performance generally requires additional FPGA resources. In contrast, area optimization often results in performance decreases. During kernel optimization, Altera recommends that you run multiple versions of the kernel on the FPGA board to determine the kernel programming strategy that generates the best size versus performance trade-off.

Compilation Considerations

You can direct the Altera Offline Compiler (AOC) to perform area usage analysis during kernel compilation.

1. To review the estimated resource usage summary on-screen, compile your kernel by including the `--report` flag in your `aoc` command. Alternatively, review the actual area usage in the **<*your_kernel_filename*>/area_report/<your_kernel_filename>.area** file.

   For more information on the .area files, refer to *Kernel-Specific Area Reports*.
2. If possible, perform floating-point computations by compiling your OpenCL kernel with the `--fpc` or `--fp-relaxed` option of the `aoc` command.

For more usage information on the `--report`, `--fp-relaxed` and `--fpc` options, refer to the *Displaying Estimated Resource Usage Summary (--report)*, *Relaxing Order of Floating-Point Operations (--fp-relaxed)*, and *Reducing Floating-Point Operations (--fpc)* sections of the *Altera SDK for OpenCL Programming Guide*.

For more information on floating-point operations, refer to *Optimize Floating-Point Operations*.

Related Information

- **Displaying the Estimated Resource Usage Summary On-Screen (--report)**
- **Relaxing the Order of Floating-Point Operations (--fp-relaxed)**
- **Reducing Floating-Point Rounding Operations (--fpc)**
- **Optimize Floating-Point Operations** on page 1-13

Kernel-Specific Area Reports

When you compilie a kernel by invoking the aoc command together with the `--report` option, the Altera Offline Compiler (AOC) displays the total estimated resource usage summary of all the kernels. The AOC also generates a variety of report files. For information on the logic utilization (that is, area usage) of a specific kernel, refer to the **<*your_kernel_filename*>.area** file in the **<*your_kernel_filename*>/area_report** directory.

Consider a **.cl** kernel source file that includes three different kernels:

```
#define N 128

__kernel void kernel1 (__global int * restrict A,
                       __global int * restrict B,
                       __global int* restrict result)
{
    int sum = 0;

    for (unsigned i=0; i < N; i++)
    {
        for (unsigned j = 0; j < N; j++)
        {
            sum += A[i*N+j];
        }
        sum += B[i];
    }

    * result = sum;
}

__kernel void kernel2 (__global const float * restrict a,
                       __global float * restrict b,
                       const float c_divided_by_d)
{
    size_t gid = get_global_id(0);
    b[gid] = a[gid] * c_divided_by_d;
}

__kernel void kernel3 (__global int * a,
                       __global int * b,
                       __global int * c)
{
    int gid = get_global_id(0);
    c[gid] = a[gid]+b[gid];
}
```

When you compile **mykernels.cl** by invoking `aoc --report mykernels.cl`, the AOC displays the following estimated resource usage summary:

```
+--------------------------------------------------------+
; Estimated Resource Usage Summary                       ;
+--------------------------------------+-----------------+
; Resource                             + Usage           ;
+--------------------------------------+ ----------------+
; Logic utilization                    ;   19%           ;
; Dedicated logic registers            ;    8%           ;
; Memory blocks                        ;   19%           ;
; DSP blocks                           ;    0%           ;
+--------------------------------------+-----------------;
```

The AOC also generates the following **.area** files in the **mykernels/area_report** directory:

kernel1.area

```
KernelTotal: LEs = 4311  FFs = 10557  RAMs = 49  DSPs = 0
LSU_resources: LEs = 1144  FFs = 2914  RAMs = 42  DSPs = 0
FP_resources: LEs = 0  FFs = 0  RAMs = 0  DSPs = 0
Local_mem_resources: LEs = 0  FFs = 0  RAMs = 0  DSPs = 0
Local_mem_ram_resources: LEs = 0  FFs = 0  RAMs = 0  DSPs = 0
Reg_State_resources: LEs = 558  FFs = 4075  RAMs = 0  DSPs = 0
RAM_State_resources: LEs = 90  FFs = 72  RAMs = 3  DSPs = 0
MrgBr_State_resources: LEs = 388  FFs = 774  RAMs = 0  DSPs = 0
Other_State_resources: LEs = 1  FFs = 1  RAMs = 0  DSPs = 0
```

```
Other_resources: LEs = 2130  FFs = 2721  RAMs = 4  DSPs = 0
------------
LEs: 0.821456 %
FFs: 1.00581 %
RAMs: 1.90884 %
DSPs: 0 %
Util: 1.79806 %
```

kernel2.area

```
KernelTotal: LEs = 3476  FFs = 5135  RAMs = 27  DSPs = 1
LSU_resources: LEs = 1478  FFs = 1871  RAMs = 27  DSPs = 0
FP_resources: LEs = 0  FFs = 0  RAMs = 0  DSPs = 0
Local_mem_resources: LEs = 0  FFs = 0  RAMs = 0  DSPs = 0
Local_mem_ram_resources: LEs = 0  FFs = 0  RAMs = 0  DSPs = 0
Reg_State_resources: LEs = 135  FFs = 1180  RAMs = 0  DSPs = 0
RAM_State_resources: LEs = 0  FFs = 0  RAMs = 0  DSPs = 0
MrgBr_State_resources: LEs = 64  FFs = 96  RAMs = 0  DSPs = 0
Other_State_resources: LEs = 164  FFs = 255  RAMs = 0  DSPs = 0
Other_resources: LEs = 1635  FFs = 1733  RAMs = 0  DSPs = 1
------------
LEs: 0.662348 %
FFs: 0.489234 %
RAMs: 1.05181 %
DSPs: 0.0509424 %
Util: 1.09585 %
```

kernel3.area

```
KernelTotal: LEs = 3670  FFs = 4755  RAMs = 40  DSPs = 0
LSU_resources: LEs = 1795  FFs = 2177  RAMs = 40  DSPs = 0
FP_resources: LEs = 0  FFs = 0  RAMs = 0  DSPs = 0
Local_mem_resources: LEs = 0  FFs = 0  RAMs = 0  DSPs = 0
Local_mem_ram_resources: LEs = 0  FFs = 0  RAMs = 0  DSPs = 0
Reg_State_resources: LEs = 29  FFs = 697  RAMs = 0  DSPs = 0
RAM_State_resources: LEs = 0  FFs = 0  RAMs = 0  DSPs = 0
MrgBr_State_resources: LEs = 64  FFs = 96  RAMs = 0  DSPs = 0
Other_State_resources: LEs = 100  FFs = 100  RAMs = 0  DSPs = 0
Other_resources: LEs = 1682  FFs = 1685  RAMs = 0  DSPs = 0
------------
LEs: 0.699314 %
FFs: 0.45303 %
RAMs: 1.55824 %
DSPs: 0 %
Util: 1.08716 %
```

Related Information

Displaying the Estimated Resource Usage Summary On-Screen (--report)

Board Variant Selection Considerations

Target a board variant in your Custom Platform that provides only the external connectivity resources you require.

For example, if your kernel requires one external memory bank, target a board variant that only supports a single external memory bank. Targeting a board with multiple external memory banks increases the area usage of your kernel unnecessarily.

If your Custom Platform does not provide a board variant that meets your needs, consider creating a board variant. Consult the *Altera SDK for OpenCL Custom Platform Toolkit User Guide* for more information.

Related Information

Altera SDK for OpenCL Custom Platform Toolkit User Guide

Memory Access Considerations

Altera recommends kernel programming strategies that can improve memory access efficiency and reduce area usage of your OpenCL kernel.

1. Minimize the number of access points to external memory.

   If possible, structure your kernel such that it reads its input from one location, processes the data internally, and then writes the output to another location.

2. Instead of relying on local or global memory accesses, structure your kernel as a single work-item with shift register inference whenever possible.
3. Instead of creating a kernel that writes data to external memory and a kernel that reads data from external memory, implement the Altera SDK for OpenCL (AOCL) channels extension between the kernels for direct data transfer.
4. If your OpenCL application includes many separate constant data accesses, declare the corresponding pointers using `__constant` instead of `__global const`. Declaration using `__global const` creates a private cache for each load or store operation. On the other hand, declaration using `__constant` creates a single constant cache on the chip only.

   Caution: If your kernel targets a Cylcone® V device (for example, Cyclone V SoC), declaring `__constant` pointer kernel arguments might degrade FPGA performance.

5. If your kernel passes a small number of constant arguments, pass them as values instead of pointers to global memory.

   For example, instead of passing `__constant int * coef` and then dereferencing `coef` with index 0 to 10, pass `coef` as a value (`int16 coef`). If `coef` was the only `__constant` pointer argument, passing it as a value eliminates the constant cache and the corresponding load and store operations completely.

6. Conditionally *shifting* large shift registers inside pipelined loops leads to the creation of inefficient hardware. For example, the following kernel consumes more resources when the `if (K > 5)` condition is present:

```
#define SHIFT_REG_LEN 1024
__kernel void bad_shift_reg (__global int * restrict src,
                             __global int * restrict dst,
                             int K)
{
    float shift_reg[SHIFT_REG_LEN];
    int sum = 0;

    for (unsigned i = 0; i < K; i++)
    {
        sum += shift_reg[0];
        shift_reg[SHIFT_REG_LEN-1] = src[i];

        // This condition will cause sever area bloat.
        if (K > 5)
        {
          #pragma unroll
          for (int m = 0; m < SHIFT_REG_LEN-1 ; m++)
          {
              shift_reg[m] = shift_reg[m + 1];
          }
```

```
            }
            dst[i] = sum;
        }
    }
```

Attention: Conditionally *accessing* a shift register does not degrade hardware efficiency. If it is necessary to implement conditional shifting of a large shift register in your kernel, consider modifying your code so that it uses local memory.

Arithmetic Operation Considerations

Select the appropriate arithmetic operation for your OpenCL application to avoid excessive FPGA area usage.

1. Introduce floating-point arithmetic operations only when necessary.
2. The Altera Offline Compiler (AOC) defaults floating-point constants to double data type. Add an `f` designation to the constant to make it a single precision floating-point operation.

   For example, the arithmetic operation `sin(1.0)` represents a double precision floating-point sine function. The arithmetic operation `sin(1.0f)` represents a single precision floating-point sine function.

3. If you do not require full precision result for a complex function, compute simpler arithmetic operations to approximate the result. Consider the following example scenarios:

   a. Instead of computing the function `pow(x,n)` where *n* is a small value, approximate the result by performing repeated squaring operations because they require much less hardware resources and area.
   b. Ensure you are aware of the original and approximated area usages because in some cases, computing a result via approximation might result in excess area usage. For example, the `sqrt` function is not resource-intensive. Other than a rough approximation, replacing the `sqrt` function with arithmetic operations that the host has to compute at runtime might result in larger area usage.
   c. If you work with a small set of input values, consider using a lookup table (LUT) instead.

4. If your kernel performs a complex arithmetic operation with a constant that the AOC computes at compilation time (for example, `log(PI/2.0)`), perform the arithmetic operation on the host instead and pass the result as an argument to the kernel at runtime.

Data Type Selection Considerations

Select the appropriate data type to optimize the FPGA area usage by your OpenCL application.

1. Select the most appropriate data type for your application.

   For example, do not define your variable as `float` if the data type `short` is sufficient.

2. Ensure that both sides of an arithmetic expression belong to the same data type.

   Consider an example where one side of an arithmetic expression is a floating-point value and the other side is an integer. The mismatched data types cause the Altera Offline Compiler (AOC) to create implicit conversion operators, which can become expensive if they are present in large numbers.

3. Take advantage of padding if it exists in your data structures.

For example, if you only need `float3` data type, which has the same size as `float4`, you may change the data type to `float4` to make use of the extra dimension to carry an unrelated value.

Additional Information

For additional information, demonstrations and training options, visit the Altera SDK for OpenCL product page on the Altera website.

Related Information

Altera SDK for OpenCL product page on the Altera website

Document Revision History

Table 1-7: Altera SDK for OpenCL Best Practices Guide Document Revision History

Date	Version	Changes
May 2015	15.0.0	• In *Memory Access Considerations*, added Caution note regarding performance degradation that might occur when declaring __constant pointer arguments in kernels targeting Cyclone® V devices. • In *Good Design Practices for Single Work-Item Kernel*, removed the *Initialize Data Prior to Usage in a Loop* section and added a *Declare Variables in the Deepest Scope Possible* section. • Added *Removing Loop-Carried Dependency by Inferring Shift Registers*. The topic discusses how, in single work-item kernels, inferring double precision floating-point array as a shift register can remove loop-carried dependencies. • Added *Kernel-Specific Area Reports* to show examples of kernel-specific **.area** files that the Altera Offline Compiler (AOC) generates during compilation. • Renamed *Transfer Data Via AOCL Channels* to *Transfer Data Via AOCL Channels or OpenCL Pipes* and added the following: ◦ More informaiton on how channels can help improve kernel performance. ◦ Information on OpenCL pipes. • Renamed *Data Type Considerations* to *Data Type Selection Considerations*.

Date	Version	Changes
December 2014	14.1.0	• Reorganized the information flow in the *Optimization Report Messages* section to update report messages and the layout of the optimization report. • Included new optimization report messages detailing the reasons for unsuccessful and suboptimal pipelined executions. • Added the *Optimization Report Messages for Simplified Analysis of a Complex Design* subsection under *Optimization Report Messages* to describe new report message for simplified kernel analysis. • Renamed *Using Feedback from the Optimization Report to Address Single Work-Item Kernels Dependencies* to *Addressing Single Work-Item Kernel Dependencies Based on Optimization Report Feedback*. • Added the *Transferring Loop-Carried Dependency to Local Memory* subsection under *Addressing Single Work-Item Kernel Dependencies Based on Optimization Report Feedback* to describe new strategy for resolving loop-carried dependency. • Updated the Resource-Driven Optimization and Compilation Considerations sections to reflect the deprecation of the `-O3` and `--util <N>` Altera Offline Compiler (AOC) command options. • Consolidated and simplified the *Heterogeneous Memory Buffers* and *Host Application Modifications for Heterogeneous Memory Accesses* sections. • Added the section *Align a Struct and Remove Padding between Struct Fields*. • Removed the section *Ensure 4-Byte Alignment to All Data Structures*. • Modified the figure *Single Work-Item Optimization Work Flow* to include emulation and profiling.

Date	Version	Changes
June 2014	14.0.0	• Renamed document as the *Altera SDK for OpenCL Best Practices Guide*. • Reorganized information flow. • Renamed *Good Design Practices* to *Good OpenCL Kernel Design Practices*. • Added channels information in *Transfer data via AOCL Channels*. • Added profiler information in *Profile Your Kernel to Identify Performance Bottlenecks*. • Added the section *Single Work-Item Kernel Versus NDRange Kernel*. • Updated *Single Work-Item Execution* section. • Removed *Performance Warning Messages* section. • Renamed *Single Work-Item Kernel Programming Considerations* to *Good Design Practices for Single Work-Item Kernel*. • Added the section *Strategies for Improving Single Work-Item Kernel Performance*. • Renamed *Optimization of Data Processing Efficiency* to *Strategies for Improving NDRange Kernel Data Processing Efficiency*. • Removed *Resource Sharing* section. • Renamed *Floating-Point Operations* to *Optimize Floating-Point Operations*. • Renamed *Optimization of Memory Access Efficiency* to *Strategies for Improving Memory Access Efficiency*. • Updated *Manual Partitioning of Global Memory* section. • Added the section *Strategies for Optimizing FPGA Area Usage*.
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*.

Date	Version	Changes
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*.

Date	Version	Changes
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.