Altera SDK for OpenCL

Programming Guide

Subscribe

Send Feedback

OCL002-15.0.0
2015.05.04

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

Contents

Altera SDK for OpenCL Programming Guide 1

2015.05.04

OCL002-15.0.0 Subscribe Send Feedback

The *Altera SDK for OpenCL Programming Guide* provides descriptions, recommendations and usage information on the Altera® Software Development Kit (SDK) for OpenCL™ (AOCL) compiler and tools. The AOCL[1] is an OpenCL[2]-based heterogeneous parallel programming environment for Altera FPGAs.

AOCL Programming Guide Prerequisites

The *Altera SDK for OpenCL Programming Guide* assumes that you are knowledgeable in OpenCL concepts and application programming interfaces (APIs). It also assumes that you have experience creating OpenCL applications and are familiar with the OpenCL Specification version 1.0.

Before using the Altera SDK for OpenCL (AOCL) or the Altera Runtime Environment (RTE) for OpenCL to program a non-SoC, familiarize yourself with the respective getting started guides. This document assumes that you have performed the following tasks:

- For the AOCL:
  - Download and install the Quartus® II software
  - Download and install the relevant device support
  - Download and install the AOCL
- For the RTE:
  - Download and install the RTE
- Install your FPGA board
- Program your non-SoC device with the hello_world example OpenCL application

If you have not performed the tasks described above, refer to the *Altera SDK for OpenCL Getting Started Guide* or the *Altera RTE for OpenCL Getting Started Guide* for more information.

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

© 2015 Altera Corporation. All rights reserved. ALTERA, ARRIA, CYCLONE, ENPIRION, MAX, MEGACORE, NIOS, QUARTUS and STRATIX words and logos are trademarks of Altera Corporation and registered in the U.S. Patent and Trademark Office and in other countries. All other words and logos identified as trademarks or service marks are the property of their respective holders as described at www.altera.com/common/legal.html. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

ISO 9001:2008 Registered

Before using the AOCL or the RTE to program a Cyclone® V SoC Development Kit, familiarize yourself with the respective getting started guides. This document assumes that you have performed the following tasks:

- For the AOCL:
  - Download and install the Quartus II software
  - Download and install the Cyclone V and Stratix® V device support files
  - Download and install the AOCL
  - Download and install the SoC Embedded Design Suite (EDS)
- For the RTE:
  - Download and install the RTE
  - Download and install the SoC Embedded Design Suite (EDS)
- Install and set up your Cyclone V SoC Development Kit
- Program your SoC with the hello_world example OpenCL application

If you have not performed the tasks described above, refer to the *Altera SDK for OpenCL Cyclone V SoC Getting Started Guide* or the *Altera RTE for OpenCL Getting Started Guide* for more information.

Related Information

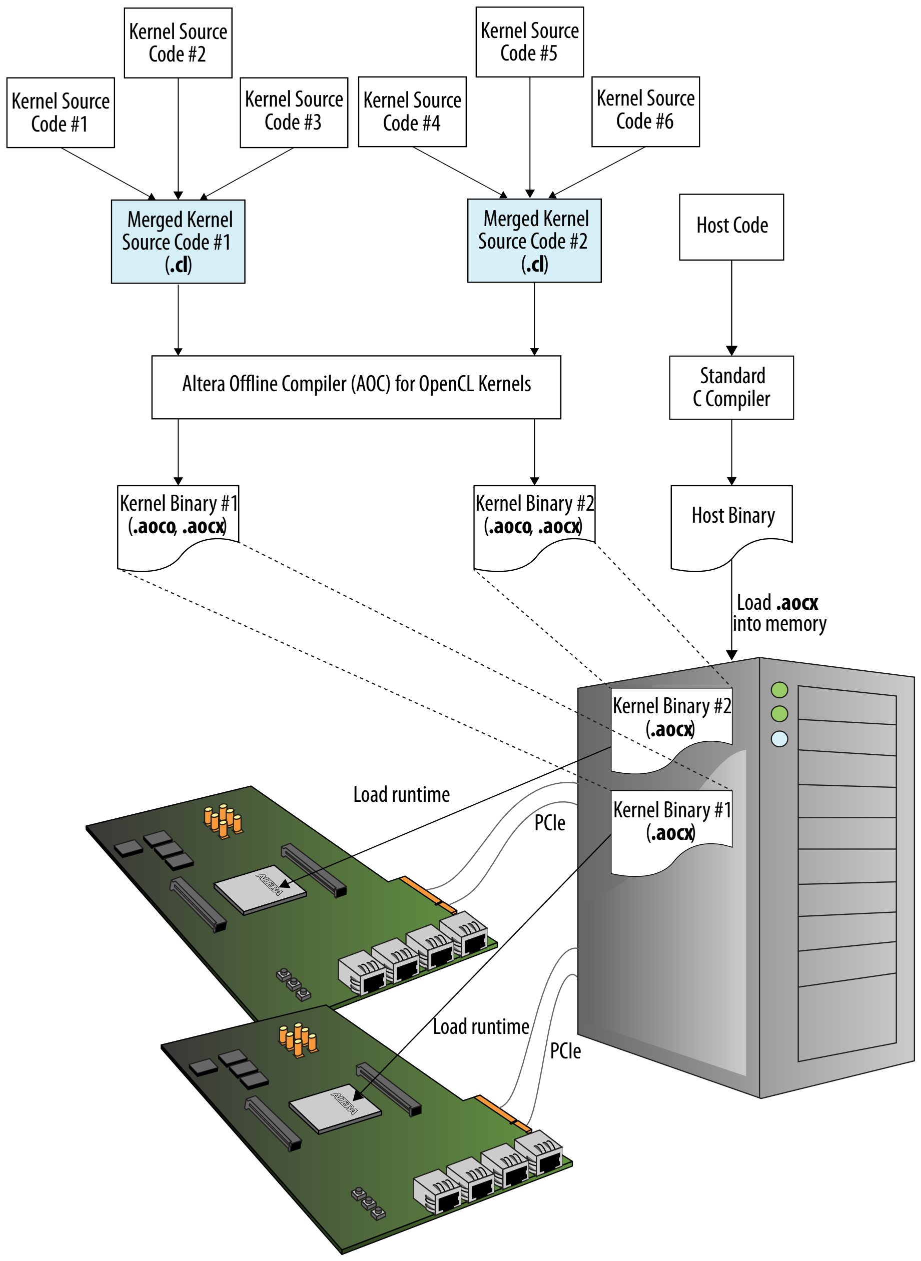
- **OpenCL References Pages**
- **OpenCL Specification version 1.0**
- **Altera SDK for OpenCL Getting Started Guide**
- **Altera RTE for OpenCL Getting Started Guide**
- **Altera SDK for OpenCL Cyclone V SoC Getting Started Guide**

AOCL FPGA Programming Flow

The Altera SDK for OpenCL (AOCL) programs an FPGA with an OpenCL application in a two-step process. The Altera Offline Compiler (AOC) first compiles your OpenCL kernels. The host-side C compiler compiles your host application and then links the compiled OpenCL kernels to it.

The following figure depicts the AOCL FPGA programming flow:

Figure 1-1: The AOCL FPGA Programming Flow



Before you compile your OpenCL kernels, you must consolidate your kernel source files into a single **.cl** source file. The OpenCL kernel source file (**.cl**) contains your OpenCL source code. The AOC compiles your kernel and generates the following files and folders:

- The *Altera Offline Compiler Object file* (**.aoco**) is an intermediate object file that contains information for later stages of the compilation.
- The *Altera Offline Compiler Executable file* (**.aocx**) is the hardware configuration file and contains information necessary at runtime.
- The ***<your_kernel_filename>*** folder or subdirectory, which contains data necessary to create the **.aocx** file.

The AOC creates the **.aocx** file from the contents of the *<your_kernel_filename>* folder or subdirectory. It also incorporates information from the **.aoco** file into the **.aocx** file during hardware compilation. The **.aocx** file contains data that the host application uses to create program objects for the target FPGA. The host application loads these program objects into memory. The host runtime then calls these program objects from memory and programs the target FPGA as required.

AOC Kernel Compilation Flows

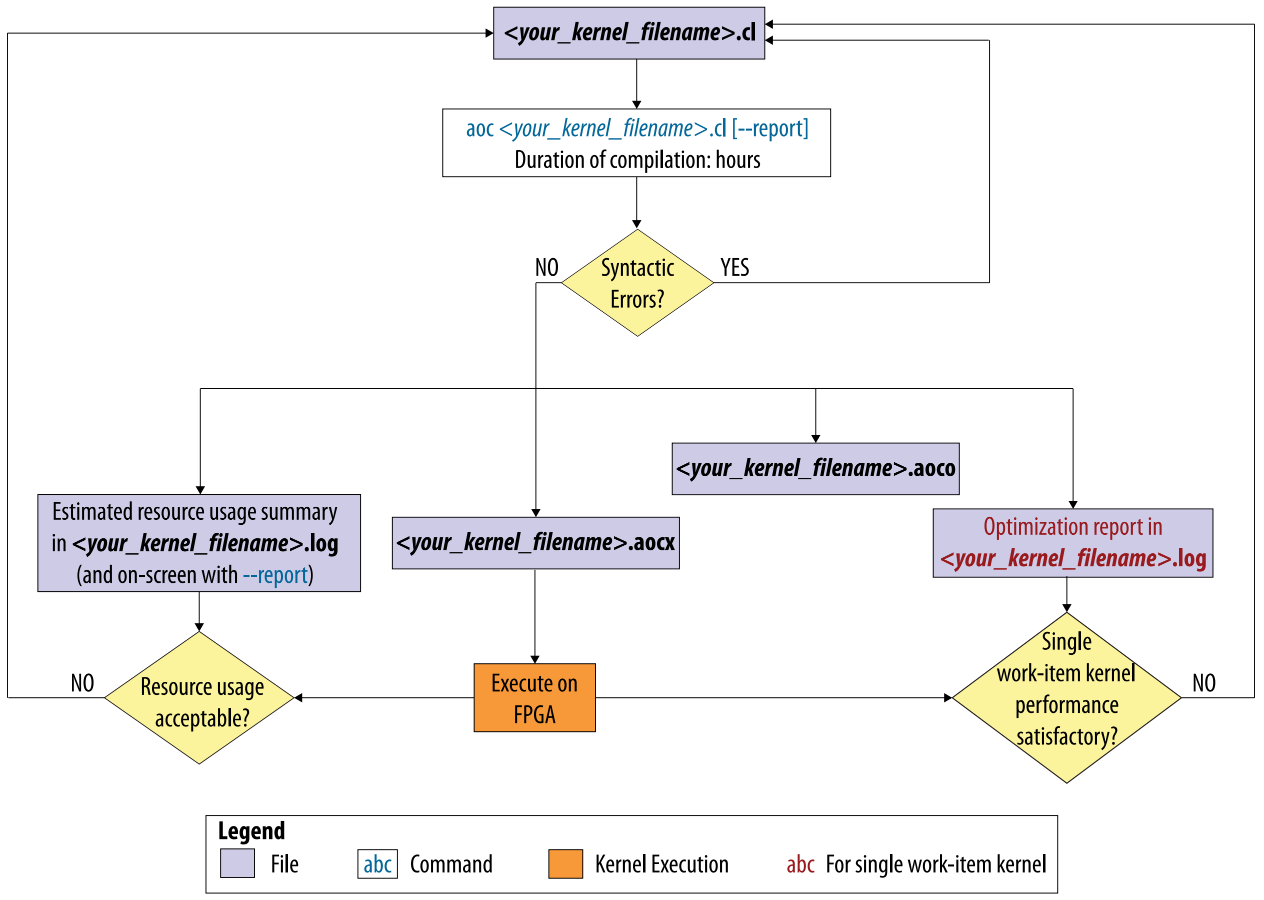
The Altera Offline Compiler (AOC) can create your FPGA hardware configuration file in a one-step or a multistep process. The complexity of your kernel dictates the AOC compilation option you implement.

One-Step Compilation for Simple Kernels

By default, the AOC compiles your OpenCL kernel and creates the hardware configuration file in a single step. Choose this compilation option only if your OpenCL application requires minimal optimizations.

The following figure illustrates the OpenCL kernel design flow that has a single compilation step.

Figure 1-2: One-Step OpenCL Kernel Compilation Flow


<your_kernel_filename>.cl
aoc <your_kernel_filename>.cl [--report]
Duration of compilation: hours
NO
Syntactic Errors?
YES
<your_kernel_filename>.aoco
Estimated resource usage summary in <your_kernel_filename>.log (and on-screen with --report)
<your_kernel_filename>.aocx
Optimization report in <your_kernel_filename>.log
NO
Resource usage acceptable?
Execute on FPGA
Single work-item kernel performance satisfactory?
NO
Legend
File
abc Command
Kernel Execution
abc For single work-item kernel

A successful compilation results in the following files and reports:

- Altera Offline Compiler Object file (**.aoco**)
- Altera Offline Compiler Executable file (**.aocx**)
- In the ***<your_kernel_filename>*/*<your_kernel_filename>*.log** file, the estimated resource usage summary provides a preliminary assessment of area usage. If you have a single work-item kernel, the optimization report identifies performance bottlenecks.

Attention: It is very time consuming to iterate on your design using the one-step compilation flow. For each iteration, you must perform a full compilation, which takes hours. Then you must execute the kernel on the FPGA before you can assess its performance.

Related Information

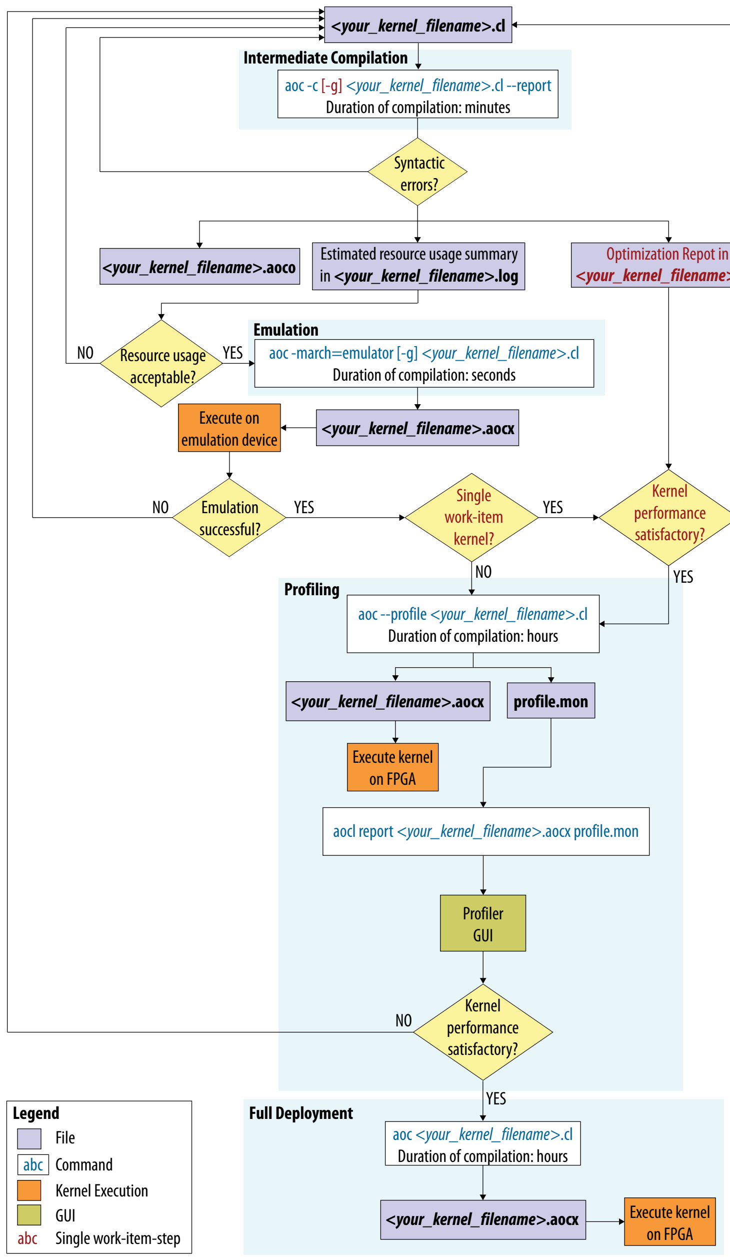
Compiling Your Kernel to Create Hardware Configuration File on page 1-76

Multistep AOCL Design Flow

Choose the multistep Altera SDK for OpenCL (AOCL) design flow if you want to iterate on your OpenCL kernel design to implement performance-improving optimizations .

The figure below outlines the stages in the AOCL design flow. The steps in the design flow serve as checkpoints for identifying functional errors and performance bottlenecks. They allow you to modify your OpenCL kernel code without performing a full compilation after each iteration.

Figure 1-3: The Multistep AOCL Design Flow


<your_kernel_filename>.cl
Intermediate Compilation
aoc -c [-g] <your_kernel_filename>.cl --report
Duration of compilation: minutes
Syntactic errors?
<your_kernel_filename>.aoco
Estimated resource usage summary in <your_kernel_filename>.log
Optimization Repot in <your_kernel_filename>.log
NO
Resource usage acceptable?
YES
Emulation
aoc -march=emulator [-g] <your_kernel_filename>.cl
Duration of compilation: seconds
Execute on emulation device
<your_kernel_filename>.aocx
NO
Emulation successful?
YES
Single work-item kernel?
YES
Kernel performance satisfactory?
NO
NO
YES
Profiling
aoc --profile <your_kernel_filename>.cl
Duration of compilation: hours
<your_kernel_filename>.aocx
profile.mon
Execute kernel on FPGA
aocl report <your_kernel_filename>.aocx profile.mon
Profiler GUI
NO
Kernel performance satisfactory?
YES
Legend
File
abc Command
Kernel Execution
GUI
abc Single work-item-step
Full Deployment
aoc <your_kernel_filename>.cl
Duration of compilation: hours
<your_kernel_filename>.aocx
Execute kernel on FPGA

The AOCL design flow includes the following steps:

1. Intermediate compilation

   The intermediate compilation step checks for syntatic errors. It then generates an Altera Offline Compiler Object file (**.aoco**) without building the hardware configuration file. The estimated resource usage summary in the ***<your_kernel_filename>*/*<your_kernel_filename>*.log** file can provide insight into the type of kernel optimizations you can perform. For a single work-item kernel, include the `-g` option to insert source information in the optimization report in the ***<your_kernel_filename>*.log** file.

2. Emulation

   Assess the functionality of your OpenCL kernel by executing it on one or multiple emulation devices on an x86-64 host. For Linux systems, include the `-g` option to enable symbolic debug support. Symbolic debug allows you to locate the origins of functional errors in your kernel code.

3. Profiling

   Instruct the AOC to instrument performance counters in the Verilog code in the Altera Offline Compiler Executable file (**.aocx**). During execution, the performance counters collect performance information which you can then review in the profiler GUI.

4. Full deployment

   If you are satisfied with the performance of your OpenCL kernel throughout the design flow, perform a full compilation. You can then execute the **.aocx** file on the FPGA.

Related Information

- **Compiling Your OpenCL Kernel** on page 1-75
- **Emulating and Debugging Your OpenCL Kernel** on page 1-84
- **Profiling Your OpenCL Kernel** on page 1-89

Obtaining General Information on Software, Compiler, and Custom Platform

The Altera SDK for OpenCL (AOCL) includes two sets of command options: the AOCL utility commands (`aocl <command_option>`) and the Altera Offline Compiler (AOC) commands (`aoc <command_option>`). Each set of commands includes options you can invoke to obtain general information on the software, the compiler, and the Custom Platform.

Displaying the Software Version (version) on page 1-8
To display the version of the Altera SDK for OpenCL (AOCL) , invoke the `version` utility command.

Displaying the Compiler Version (--version) on page 1-8
To display the version of the Altera Offline Compiler (AOC), invoke the `version` compiler command.

Listing the AOCL Utility Command Options (help) on page 1-8
To display information on the Altera SDK for OpenCL (AOCL) utility command options, invoke the `help` utility command.

Listing the AOC Command Options (-h or --help) on page 1-8
To display information on the Altera Offline Compiler (AOC) command options, invoke the `help` or `h` compiler command.

Listing the Available FPGA Boards in Your Custom Platform (--list-boards) on page 1-9
To list the FPGA boards available in your Custom Platform, include the `--list-boards` option in the `aoc` command.

Displaying the Software Version (version)

To display the version of the Altera SDK for OpenCL (AOCL) , invoke the `version` utility command.

- At the command prompt, invoke the `aocl version` command.
  Example output:

```
aocl <version>.<build> (Altera SDK for OpenCL, Version <version>
Build <build>, Copyright (C) 2014 Altera Corporation)
```

Displaying the Compiler Version (--version)

To display the version of the Altera Offline Compiler (AOC), invoke the `version` compiler command.

- At a command prompt, invoke the `aoc --version` command.
  Example output:

```
Altera SDK for OpenCL, 64-Bit Offline Compiler
Version <version> Build <build>
Copyright (C) <year> Altera Corporation
```

Listing the AOCL Utility Command Options (help)

To display information on the Altera SDK for OpenCL (AOCL) utility command options, invoke the `help` utility command.

- At a command prompt, invoke the `aocl help` command.
  The AOCL categorizes the utility command options based on their functions. It also provides a description for each option.

Displaying Information on an AOCL Utility Command Option (help <command_option>)

To display information on a specific Altera SDK for OpenCL (AOCL) utility command option, include the subcommand as an argument of the `help` utility command.

- At a command prompt, invoke the `aocl help <command_option>` command.
  For example, to obtain more information on the `install` utility command option, invoke the `aocl help install` command.
  Example output:

```
aocl install - Installs a board onto your host system.

Usage: aocl install

Description:
This command installs a board's drivers and other necessary software for the
host operating system to communicate with the board.
For example this might install PCIe drivers.
```

Listing the AOC Command Options (-h or --help)

To display information on the Altera Offline Compiler (AOC) command options, invoke the `help` or `h` compiler command.

- At a command prompt, invoke the `aoc --help` or `aoc -h` command.
  The Altera SDK for OpenCL (AOCL) categorizes the AOC command options based on their functions. It also provides a description for each option.

Listing the Available FPGA Boards in Your Custom Platform (--list-boards)

To list the FPGA boards available in your Custom Platform, include the `--list-boards` option in the `aoc` command.

Before you begin

To view the list of available boards in your Custom Platform, you must first set the environment variable *AOCL_BOARD_PACKAGE_ROOT* to point to the location of your Custom Platform.

- At a command prompt, invoke the `aoc --list-boards` command.
  The Altera Offline Compiler (AOC) generates an output that resembles the following:

```
Board list:
   <board_name_1>
   <board_name_2>
...
```

  where <*board_name_N*> is the board name you use in your `aoc` command to target a specific FPGA board.

Managing an FPGA Board

The Altera SDK for OpenCL (AOCL) includes utility commands you can invoke to install, uninstall, diagnose, and program your FPGA board.

Installing an FPGA Board (install) on page 1-9
To install your board into the host system, invoke the `install` utility command.

Uninstalling the FPGA Board (uninstall) on page 1-11
To uninstall an FPGA board, invoke the `uninstall` utility command, uninstall the Custom Platform, and unset the relevant environment variables.

Querying the Device Name of Your FPGA Board (diagnose) on page 1-11
When you query a list of accelerator boards, the AOCL produces a list of installed devices on your machine in the order of their device names.

Running a Board Diagnostic Test (diagnose <device_name>) on page 1-12
To perform a detailed diagnosis on a specific FPGA board, include <*device_name*> as an argument of the `diagnose` utility command.

Programming the FPGA Offline or without a Host (program <device_name>) on page 1-12
To program an FPGA device offline or without a host, invoke the `program` utility command.

Programming the Flash Memory (flash <device_name>) on page 1-12
If supported, invoke the `flash` utility command to initialize the FPGA with a specified startup configuration.

Installing an FPGA Board (install)

Before creating an OpenCL application for an FPGA boards, you must first download and install the Custom Platform from your board vendor. Most Custom Platform installers require administrator privileges. To install your board into the host system, invoke the `install` utility command.

The steps below outline the board installation procedure. Some Custom Platforms require additional installation tasks. Consult your board vendor's documentation for further information on board installation.

Attention: If you are installing the Cyclone V SoC Development Kit for use with the Cyclone V SoC Development Kit Reference Platform, refer to *Installing the Cyclone V SoC Development Kit* in the *Altera SDK for OpenCL Cyclone V SoC Getting Started Guide* for more information.

1. Follow your board vendor's instructions to connect the FPGA board to your system.
2. Download the Custom Platform for your FPGA board from your board vendor's website.

   For more information, refer to the OpenCL Reference Platforms page on the Altera website.
3. Install the Custom Platform in a directory that you own (that is, not a system directory).
4. Set the environment variable *AOCL_BOARD_PACKAGE_ROOT* to point to the location of the Custom Platform subdirectory containing the **board_env.xml** file.

   For example, for the Stratix V Network Reference Platform (s5_net), set *AOCL_BOARD_PACKAGE_ROOT* to point to the ***<path_to_s5_net>*/s5_net** directory.

5. Add the Custom Platform library paths to the *PATH* (Windows) or *LD_LIBRARY_PATH* (Linux) environment variable setting. You may apply permanent settings manually by adding the path to the memory-mapped (MMD) library within the Custom Platform. Alternatively, you may apply transient settings to the current session by running the ***ALTERAOCLSDKROOT*/init_opencl** script.

   For example, if you use s5_net, the Windows *PATH* environment variable setting is **%*AOCL_BOARD_PACKAGE_ROOT*%\windows64\bin**. The Linux *LD_LIBRARY_PATH* setting is **$*AOCL_BOARD_PACKAGE_ROOT*/linux64/lib**.

   The *Altera SDK for OpenCL Getting Started Guide* contains more information on the init_opencl script. For information on **init_opencl.bat**, refer to the *Setting the Environment Variables for Windows* section. For information on **init_opencl.sh**, refer to the *Setting the Environment Variables for Linux* section.
6. Invoke the command `aocl install` at a command prompt.

   Invoking `aocl install` also installs a board driver that allows communication between host applications and hardware kernel programs.
7. To query a list of FPGA devices installed in your machine, invoke the `aocl diagnose` command. The software generates an output that includes the <*device_name*>, which is an acl number that ranges from acl0 to acl15.

   For more information on querying the <*device_name*> of your accelerator board, refer to the *Querying the Device Name of Your FPGA Board* section.

8. To verify the successful installation of the FPGA board, invoke the command `aocl diagnose <device_name>` to run any board vendor-recommended diagnostic test.

Related Information

- Installing the Cyclone V SoC Development Kit
- Querying the Device Name of Your FPGA Board (diagnose) on page 1-11
- Setting the Environment Variables for Windows
- Setting the Environment Variables for Linux

Uninstalling the FPGA Board (uninstall)

To uninstall an FPGA board, invoke the `uninstall` utility command, uninstall the Custom Platform, and unset the relevant environment variables. You must uninstall the existing FPGA board if you migrate your OpenCL application to another FPGA board from a different Custom Platform.

To uninstall your FPGA board, perform the following tasks:

1. Following your board vendor's instructions to disconnect the board from your machine.
2. Invoke the `aocl uninstall` utility command to remove the current host computer drivers (for example, PCI Express® (PCIe®) drivers). The AOCL uses these drivers to communicate with the FPGA board.
3. Uninstall the Custom Platform.
4. Unset the *LD_LIBRARY_PATH* (for Linux) or *PATH* (for Windows) environment variable.
5. Unset the *AOCL_BOARD_PACKAGE_ROOT* environment variable.

Querying the Device Name of Your FPGA Board (diagnose)

Some AOCL utility commands require you to specify the device name (*<device_name>*). The *<device_name>* refers to the acl number (e.g. acl0 to acl15) that corresponds to the FPGA device. When you query a list of accelerator boards, the AOCL produces a list of installed devices on your machine in the order of their device names.

- To query a list of installed devices on your machine, type `aocl diagnose` at a command prompt. The software generates an output that resembles the example shown below:

```
aocl diagnose: Running diagnostic from ALTERAOCLSDKROOT/board/<board_name>/
<platform>/libexec

Verified that the kernel mode driver is installed on the host machine.

Using board package from vendor: <board_vendor_name>
Querying information for all supported devices that are installed on the host
machine ...

device_name  Status  Information

acl0         Passed  <descriptive_board_name>
                     PCIe dev_id = <device_ID>, bus:slot.func = 02:00.00,
                       at Gen 2 with 8 lanes.
                     FPGA temperature=43.0 degrees C.

acl1         Passed  <descriptive_board_name>
                     PCIe dev_id = <device_ID>, bus:slot.func = 03:00.00,
                       at Gen 2 with 8 lanes.
                     FPGA temperature = 35.0 degrees C.

Found 2 active device(s) installed on the host machine, to perform a full
diagnostic on a specific device, please run aocl diagnose <device_name>

DIAGNOSTIC_PASSED
```

Related Information

Probing the OpenCL FPGA Devices on page 1-71

Running a Board Diagnostic Test (diagnose <device_name>)

To perform a detailed diagnosis on a specific FPGA board, include <*device_name*> as an argument of the `diagnose` utility command.

- At a command prompt, invoke the `aocl diagnose <device_name>` command, where <*device_name*> is the acl number (for example, acl0 to acl15) that corresponds to your FPGA device.

  You can identify the <*device_name*> when you query the list of installed boards in your system.

Consult your board vendor's documentation for more board-specific information on using the `diagnose` utility command to run diagnostic tests on multiple FPGA boards.

Programming the FPGA Offline or without a Host (program <device_name>)

To program an FPGA device offline or without a host, invoke the `program` utility command.

- At a command prompt, invoke the `aocl program <device_name> <your_kernel_filename>.aocx` command

  where:

  <*device_name*> refers to the acl number (for example, acl0 to acl15) that corresponds to your FPGA device, and

  ***<your_kernel_filename>*.aocx** is the Altera Offline Compiler Executable file you use to program the hardware.

Programming the Flash Memory (flash <device_name>)

If supported, invoke the `flash` utility command to initialize the FPGA with a specified startup configuration.

Note: For instructions on programming the micro SD flash card of the Cyclone V SoC Development Kit, refer to the *Writing an SD Card Image onto the Micro SD Flash Card* sections for Windows and Linux in the Altera SDK for OpenCL Cyclone V SoC Getting Started Guide.

- At a command prompt, invoke the `aocl flash <device_name> <your_kernel_filename>.aocx` command

  where:

  <*device_name*> refers to the acl number (for example, acl0 to acl15) that corresponds to your FPGA device, and

  ***<your_kernel_filename>*.aocx** is the Altera Offline Compiler Executable file you use to program the hardware.

Related Information

- **Writing an SD Card Image onto the Micro SD Flash Card on Windows**
- **Writing an SD Card Image onto the Micro SD Flash Card on Linux**

Structuring Your OpenCL Kernel

Altera offers recommendations on how to structure your OpenCL kernel code. Consider implementing these programming recommendations when you create a kernel or modify a kernel written originally to target another architecture.

Guidelines for Naming the Kernel

Altera recommends that you include only alphanumeric characters in your filenames.

1. Begin a filename with an alphanumeric character.

   If the filename of your OpenCL application begins with a nonalphanumeric character, compilation fails with the following error message:

   ```
   Error: Quartus compilation FAILED
   See quartus_sh_compile.log for the output log.
   ```

2. Do not differentiate filenames using nonalphanumeric characters.

The Altera Offline Compiler (AOC) translates any nonalphanumeric character into an underscore ("_"). If you differentiate two filenames by ending them with different nonalphanumeric characters only (for example, **myKernel#.cl** and **myKernel&.cl**), the AOC translates both filenames to **<*your_kernel_filename*>_.cl** (for example, **myKernel_.cl**).

3. For Windows system, ensure that the combined length of the kernel filename and its file path does not exceed 260 characters.

   64-bit Windows 7 has a 260-character limit on the length of a file path. If the combined length of the kernel filename and its file path exceeds 260 characters, the AOC generates the following error message:

   ```
   The filename or extension is too long.
   The system cannot find the path specified.
   ```

   In addition to the AOC error message, the following error message appears in the **<*your_kernel_filename*>/quartus_sh_compile.log** file:

   ```
   Error: Can't copy <file_type> files: Can't open
   <your_kernel_filename> for write: No such file or directory
   ```

4. Do not name your **.cl** OpenCL kernel source file "kernel". Naming the source file **kernel.cl** causes the AOC to generate intermediate design files that have the same names as certain internal files, which leads to an compilation error.

Programming Strategies for Optimizing Data Processing Efficiency

Optimize the data processing efficiency of your kernel by implementing strategies such as unrolling loops, setting work-group sizes, and specifying compute units and work-items.

Unrolling a Loop

The Altera Offline Compiler (AOC) might unroll simple loops even if they are not annotated by a pragma. To direct the AOC to unroll a loop, insert an `unroll` kernel pragma in the kernel code preceding a loop you wish to unroll.

Attention:

- Provide an unroll factor whenever possible. To specify an unroll factor *N*, insert the `#pragma unroll <N>` directive before a loop in your kernel code.

  The AOC attempts to unroll the loop at most <*N*> times.

  Consider the code fragment below. By assigning a value of 2 as an argument to `#pragma unroll`, you direct the AOC to unroll the loop twice.

  ```
  #pragma unroll 2
  for(size_t k = 0; k < 4; k++)
  {
     mac += data_in[(gid * 4) + k] * coeff[k];
  }
  ```

- To unroll a loop fully, you may omit the unroll factor by simply inserting the `#pragma unroll` directive before a loop in your kernel code.

  The AOC attempts to unroll the loop fully if it understands the trip count. The AOC issues a warning if it cannot execute the unroll request.

Specifying Work-Group Sizes

Specify a maximum or required work-group size whenever possible. The Altera Offline Compiler (AOC) relies on this specification to optimize hardware usage of the OpenCL kernel without involving excess logic.

If you do not specify a `max_work_group_size` or a `reqd_work_group_size` attribute in your kernel, the work-group size assumes a default value depending on compilation time and runtime constraints.

- If your kernel contains a barrier, the AOC sets a default maximum work-group size of 256 work-items.
- If your kernel contains a barrier or refers to the local work-item ID, or if you query the work-group size in your host code, the runtime defaults the work-group size to one work-item.
- If your 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 runtime defaults the work-group size to the global NDRange size.

To specify the work-group size, modify your kernel code in the following manner:

- To specify the maximum number of work-items the AOC may allocate to a work-group in a kernel, insert the `max_work_group_size(X, Y, Z)` attribute in your kernel source code.
  For example:

```
__attribute__((max_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 specify the required number of work-items the AOC allocates to a work-group in a kernel, insert the `reqd_work_group_size(X, Y, Z)` attribute to your kernel source code.
  For example:

```
__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 AOC allocates the exact amount of hardware resources to manage the work-items in a work-group.

Specifying Number of Compute Units

To increase the data-processing efficiency of an OpenCL kernel, you can instruct the Altera Offline Compiler (AOC) to generate multiple kernel compute units, each capable of executing multiple work-groups simultaneously.

Caution: Multiplying the number of kernel compute units increases data throughput at the expense of global memory bandwidth contention among compute units.

- To specify the number of compute units for a kernel, insert the `num_compute_units(N)` attribute in the kernel source code.
  For example, the code fragment below directs the AOC to instantiate two compute units in a kernel:

```
__attribute__((num_compute_units(2)))
__kernel void test(__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 AOC distributes work-groups across the specified number of compute units.

Specifying Number of SIMD Work-Items

To increase the data-processing efficiency of an OpenCL kernel, specify the number of work-items within a work-group that the Altera Offline Compiler (AOC) executes in a single instruction multiple data (SIMD) manner.

Important: Introduce the `num_simd_work_items` attribute in conjunction with the `reqd_work_group_size` attribute. The `num_simd_work_items` attribute you specify must evenly divides the work-group size you specify for the `reqd_work_group_size` attribute.

- To specify the number of SIMD work-items in a work-group, insert the `num_simd_work_item(N)` attribute in the kernel source code.
  For example, the code fragment below assigns a fixed work-group size of 64 work-items to a kernel. It then consolidates the work-items within each work-group into four SIMD vector lanes:

```
__attribute__((num_simd_work_items(4)))
__attribute__((reqd_work_group_size(64,1,1)))
__kernel void test(__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 AOC replicates the kernel datapath according to the value you specify for `num_simd_work_items` whenever possible.

Programming Strategies for Optimizing Memory Access Efficiency

Optimize the memory access efficiency of your kernel by implementing strategies such as specifying local memory pointer size and specifying global memory buffer location.

Specifying Pointer Size in Local Memory

Optimize local memory hardware footprint (that is, size) by specifying a pointer size in bytes.

- To specify a pointer size other than the default size of 16 kilobytes (kB), include the `local_mem_size(N)` attribute in the pointer declaration within your kernel source code.
  For example:

  ```
  __kernel void myLocalMemoryPointer(
     __local float * A,
     __attribute__((local_mem_size(1024))) __local float * B,
     __attribute__((local_mem_size(32768))) __local float * C)
  {
     //statements
  }
  ```

  In the `myLocalMemoryPointer` kernel, 16 kB of local memory (default) is allocated to pointer `A`, 1 kB is allocated to pointer `B`, and 32 kB is allocated to pointer `C`.

Specifying Buffer Location in Global Memory

Specify the global memory type to which the host allocates a buffer.

1. Determine the names of the global memory types available on your FPGA board in the following manners:
   - Refer to the board vendor's documentation for more information.
   - Find the names in the **board_spec.xml** file of your board Custom Platform. For each global memory type, the name is the unique string assigned to the `name` attribute of the `global_mem` element.
2. To instruct the host to allocate a buffer to a specific global memory type, insert the `buffer_location("<memory_type>")` attribute, where *<memory_type>* is the name of the global memory type provided by your board vendor.
   For example:

   ```
   __kernel void foo(__global __attribute__((buffer_location("DDR"))) int *x,
                     __global __attribute__((buffer_location("QDR"))) int *y)
   ```

   If you do not specify the `buffer_location` attribute, the host allocates the buffer to the default memory type automatically. To determine the default memory type, consult the documentation provided by your board vendor. Alternatively, in the **board_spec.xml** file of your Custom Platform, search for the memory type that is defined first or has the attribute `default=1` assigned to it.

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

```
#define QDR\
    __global\
    __attribute__((buffer_location("QDR")))

#define DDR\
    __global\
    __attribute__((buffer_location("DDR")))

__kernel void foo (QDR uint * data, DDR uint * lup)
{
    //statements
}
```

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

Implementing AOCL Channels Extension

The Altera SDK for OpenCL (AOCL) channels extension provides a mechanism for passing data to kernels and synchronizing kernels with high efficiency and low latency.

Attention: If you want to leverage the capabilities of channels but have the ability to run your kernel program using other SDKs, implement OpenCL pipes instead.

Related Information

Implementing OpenCL Pipes on page 1-35

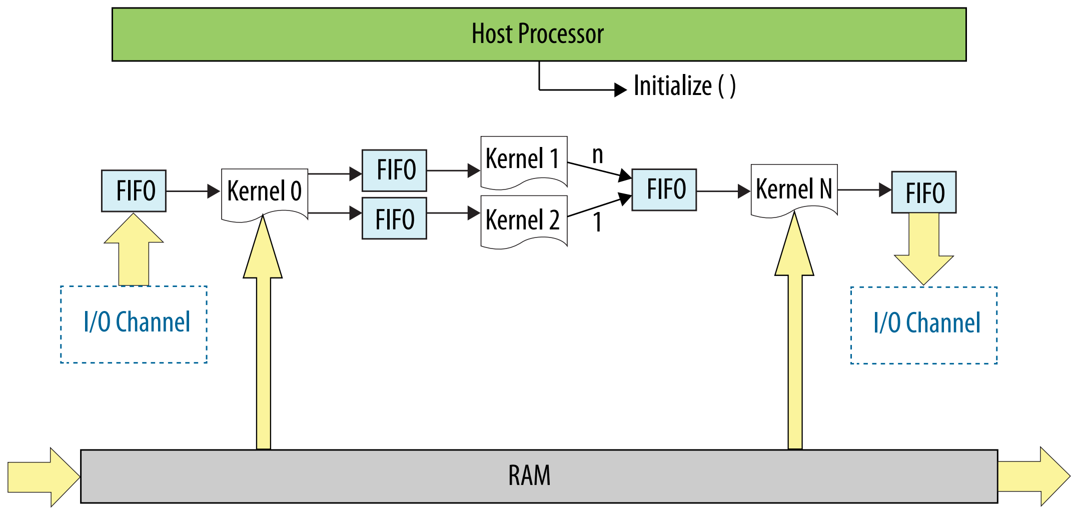
Overview of the AOCL Channels Extension

The Altera SDK for OpenCL (AOCL) channels extension allows kernels to communicate directly with each other via FIFO buffers.

Implementation of channels decouples kernel execution from the host processor. Unlike the typical OpenCL execution model, the host does not need to coordinate data movement across kernels.

The following figure provides an overview of the implementation of channels.

Figure 1-4: AOCL Channels Example



Channel Data Behavior

Data written to a channel remains in a channel as long as the kernel program remains loaded on the FPGA device. In other words, data written to a channel persists across multiple work-groups and NDRange invocations. However, data is not persistent across multiple or different invocations of kernel programs.

Consider the following code example:

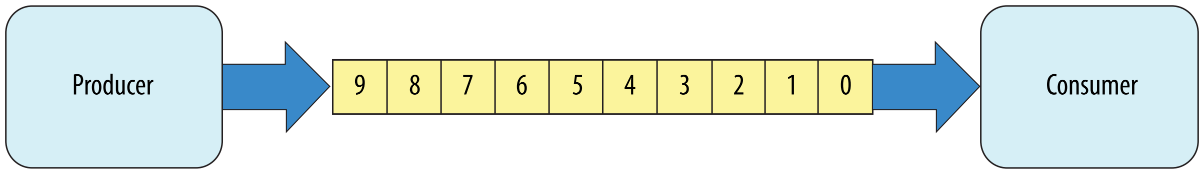
```
#pragma OPENCL EXTENSION cl_altera_channels : enable
channel int c0;
```

```
__kernel void producer()
{
    for(int i=0; i < 10; i++)
    {
        write_channel_altera(c0, i);
    }
}

__kernel void consumer( __global uint * restrict dst )
{
    for(int i=0; i < 5; i++)
    {
        dst[i] = read_channel_altera(c0);
    }
}
```

The figure below illustrates the order in which the `producer` kernel writes the elements to the channel:

Figure 1-5: Channel Data FIFO Ordering



The kernel `producer` writes ten elements ([0, 9]). The kernel `consumer` reads five elements from the channel per NDRange invocation. During the first invocation, the kernel `consumer` reads values 0 to 4 from the channel. Because the data persists across NDRange invocations, the second time you execute the kernel `consumer`, it reads values 5 to 9.

For this example, to avoid a deadlock from occurring, you need to invoke the kernel `consumer` twice for every invocation of the kernel `producer`. If you call `consumer` less than twice, `producer` stalls because the channel becomes full. If you call `consumer` more than twice, `consumer` stalls because there is insufficient data in the channel.

Multiple Work-Item Ordering for Channels

The OpenCL specification does not define a work-item ordering. The Altera SDK for OpenCL (AOCL) enforces a work-item order to maintain the consistency in channel read and write operations.

Multiple work-item accesses to a channel can be useful in some scenarios. For example, they are useful when data words in the channel are independent, or when the channel is implemented for control logic. The main concern regarding multiple work-item accesses to a channel is the order in which the kernel writes data to and reads data from the channel. If possible, the AOCL channels extension processes work-items read and write operations to the channel in a deterministic order. As such, the read and write operations remain consistent across kernel invocations.

Requirements for Deterministic Multiple Work-Item Ordering

To guarantee deterministic ordering, the AOCL checks that the channel call is work-item invariant based on the following characteristics:

1. Work-items must pass through a channel call before exiting the kernel function.
2. Work-items must pass through a channel call before entering the ensuing kernel function.
3. If either of the preceding characteristics is not satisfied, the AOCL checks that all branch conditions to the channel call basic block is work-item invariant.

If the AOCL cannot guarantee deterministic ordering of multiple work-item accesses to a channel, it warns you that the channels might not have well-defined ordering with nondeterministic execution. Primarily, the AOCL fails to provide deterministic ordering if you have work-item-variant code on loop executions with channel calls, as illustrated below:

```
__kernel void ordering( __global int * restrict check,
                        __global int * restrict data )
{
    int condition = check[get_global_id(0)];

    if(condition)
    {
        for(int i=0; i < N, i++)
        {
            process(data);
            write_channel_altera(req, data[i]);
        }
    }
    else
    {
        process(data);
    }
}
```

Because the Altera Offline Compiler (AOC) performs many transformations, such as branch conversion, during kernel invocations, it might be difficult to determine if the requirements are fulfilled for a given channel call. The AOCL generates a graphical report on channel connectivity across multiple kernels.

Work-Item Serial Execution of Channels

Work-item serial execution refers to an ordered execution behavior where work-item sequential IDs determine their execution order in the compute unit.

When you implement channels in a kernel, the Altera Offline Compiler (AOC) enforces that kernel behavior is equivalent to having at most one work-group in flight. The AOC also ensures that the kernel executes channels in work-item serial execution, where the kernel executes work-items with smaller IDs first. A work-item has the identifier `(x, y, z, group)`, where `(x, y, z)` is the local 3D identifier, and `group` is the work-group identifier.

The work-item ID `(x0, y0, z0, group0)` is considered to be smaller than the ID `(x1, y1, z1, group1)` if one of the following conditions is true:

- `group0` < `group1`
- `group0` = `group1` and `z0` < `z1`
- `group0` = `group1` and `z0` = `z1` and `y0` < `y1`
- `group0` = `group1` and `z0` = `z1` and `y0` = `y1` and `x0` < `x1`

For example, the work-item with an ID `(x0, y0, z0, group0)` executes the write channel call first, and then the work-item with an ID `(x1, y0, z0, group0)` executes the call, and so on, in a sequential order. Defining this order ensures that the system is verifiable with external models.

Channel Execution in Loop with Multiple Work-Items

When channels exist in the body of a loop with multiple work-items, as shown below, each loop iteration executes prior to subsequent iterations. This implies that loop iteration 0 of each work-item in a work-group executes before iteration 1 of each work-item in a work-group, and so on.

```
__kernel void ordering( __global int * data )
{
    write_channel_altera(req, data[get_global_id(0)]);
}
```

AOCL Channels Extension: Restrictions

There are certain design restrictions to the implementation of channels in your OpenCL application.

Single Call Site

Because the channel read and write operations do not function deterministically, for a given kernel, you can only assign one call site per channel ID. For example, the Altera Offline Compiler (AOC) cannot compile the following code example:

```
in_data1 = read_channel_altera(channel1);
in_data2 = read_channel_altera(channel2);
in_data3 = read_channel_altera(channel1);
```

The second `read_channel_altera` call to `channel1` causes compilation failure because it creates a second call site to `channel1`.

To gather multiple data from a given channel, divide the channel into multiple channels, as shown below:

```
in_data1 = read_channel_altera(channel1);
in_data2 = read_channel_altera(channel2);
in_data3 = read_channel_altera(channel3);
```

Because you can only assign a single call site per channel ID, you cannot unroll loops containing channels. Consider the following code:

```
#pragma unroll 4
for (int i=0; i < 4; i++)
{
    in_data = read_channel_altera(channel1);
}
```

The AOC issues the following warning message during compilation:

```
Compiler Warning: Unroll is required but the loop cannot be unrolled.
```

Feedback and Feed-Forward Channels

Channels within a kernel can be either `read_only` or `write_only`. Performance of a kernel that reads and writes to the same channel is poor.

Static Indexing

The Altera SDK for OpenCL (AOCL) channels extension does not support dynamic indexing into arrays of channel IDs.

Consider the following example:

```
#pragma OPENCL EXTENSION cl_altera_channels : enable

channel int ch[WORKGROUP_SIZE];

__kernel void consumer()
{

    int gid = get_global_id(0);
    int value = read_channel_altera(ch[gid]);

    //statements
}
```

Compilation of this example kernel fails with the following error message:

```
Compiler Error: Indexing into channel array ch could not be resolved to all constant
```

To avoid this compilation error, index into arrays of channel IDs statically, as shown below:

```
#pragma OPENCL EXTENSION cl_altera_channels : enable

channel int ch[WORKGROUP_SIZE];

__kernel void consumer()
{

    int gid = get_global_id(0);
    int value;

    switch(gid)
    {
        case0: value = read_channel_altera(ch[gid]); break;
        case1: value = read_channel_altera(ch[gid]); break;
        case2: value = read_channel_altera(ch[gid]); break;
        case3: value = read_channel_altera(ch[gid]); break;
        //statements

        case WORKGROUP_SIZE-1:read_channel_altera(channel[WORKGROUP_SIZE-1]); break;
    }

    //statements
}
```

Kernel Vectorization Support

You cannot vectorize kernels that use channels; that is, do not include the `num_simd_work_items` kernel attribute in your kernel code. Vectorizing a kernel that uses channels creates multiple channel masters and requires arbitration, which the AOCL channels extension does not support.

Instruction-Level Parallelism on read_channel_altera and write_channel_altera Calls

If no data dependencies exist between `read_channel_altera` and `write_channel_altera` calls, the AOC attempts to execute these instructions in parallel. As a result, the AOC might execute these read_channel_altera and write_channel_altera calls in an order that does not follow the sequence expressed in the OpenCL kernel code.

Consider the following code sequence:

```
in_data1 = read_channel_altera(channel1);
in_data2 = read_channel_altera(channel2);
in_data3 = read_channel_altera(channel3);
```

Because there are no data dependencies between the read_channel_altera calls, the AOC can execute them in any order.

Enabling the AOCL Channels for OpenCL Kernel

To implement the Altera SDK for OpenCL (AOCL) channels extension, modify your OpenCL kernels to include channels-specific pragma and application programming interface (API) calls.

Channel declarations are unique within a given OpenCL kernel program. Also, channel instances are unique for every OpenCL kernel program device pair. If the runtime loads a single OpenCL kernel program onto multiple devices, each device will have a single copy of the channel. However, these channel copies are independent and do not share data across the devices.

Declaring the Channels OPENCL EXTENSION pragma

To enable the Altera SDK for OpenCL (AOCL) channels extension, declare the OPENCL EXTENSION pragma for channels at the beginning of your kernel source code.

- To enable the AOCL channels extension, include the following line in your kernel source code to declare the OPENCL EXTENSION pragma:
  `#pragma OPENCL EXTENSION cl_altera_channels : enable`

Declaring the Channel Handle

Use the channel variable to define the connectivity between kernels or between kernels and I/O.

To read from and write to a channel, the kernel must pass the channel variable to each of the corresponding application programming interface (API) call.

- Declare the channel handle as a file scope variable in the kernel source code in the following convention: channel <*type*> <*variable_name*>
  For example: channel int c;
- The AOCL channel extension supports simultaneous channel accesses by multiple variables declared in a data structure. Declare a struct data structure for a channel in the following manner:

```
typedef struct type_ {
    int a;
    int b;
} type_t;

channel type_t foo;
```

Implementing Blocking Channel Write Extensions

The write_channel_altera application programming interface (API) call allows you to send data across a channel.

Note: The write channel calls support single-call sites only. For a given channel, only one write channel call to it can exist in the entire kernel program.

- To implement a blocking channel write, include the following `write_channel_altera` function signature:
  `void write_channel_altera (channel <type> channel_id, const <type> data);`

  Where:

  `channel_id` identifies the buffer to which the channel connects, and it must match the `channel_id` of the corresponding read channel (`read_channel_altera`).

  `data` is the data that the channel write operation writes to the channel. Data <*type*> must match the <*type*> of the `channel_id`.

  <*type*> defines a channel data width, which cannot be a constant. Follow the OpenCL conversion rules to ensure that data the kernel writes to a channel is convertible to <*type*>.

The following code snippet demonstrates the implementation of the `write_channel_altera` API call:

```
//Enables the channels extension.
#pragma OPENCL EXTENSION cl_altera_channels : enable

//Defines chan, the kernel file-scope channel variable.
channel long chan;

/*Defines the kernel which reads eight bytes (size of long) from global
memory, and passes this data to the channel.*/
__kernel void kernel_write_channel( __global const long * src )
{
    for(int i=0; i < N; i++)
    {
        //Writes the eight bytes to the channel.
        write_channel_altera(chan, src[i]);
    }
}
```

Caution: When you send data across a write channel using the `write_channel_altera` API call, keep in mind that if the channel is full (that is, if the FIFO buffer is full of data), your kernel will stall. Use the Altera SDK for OpenCL (AOCL) Profiler to check for channel stalls.

Related Information

Profiling Your OpenCL Kernel on page 1-89

Implementing Nonblocking Channel Write Extensions

Perform nonblocking channel writes to facilitate applications where data write operations might not occur. A nonblocking channel write extension returns a Boolean value that indicates whether data is written to the channel.

Consider a scenario where your application has one data producer with two identical workers. Assume the time each worker takes to process a message varies depending on the contents of the data. In this case, there might be situations where one worker is busy while the other is free. A nonblocking write can facilitate work distribution such that both workers are busy.

- To implement a nonblocking channel write, include the following `write_channel_nb_altera` function signature:
  `bool write_channel_nb_altera(channel <type> channel_id, const <type> data);`

The following code snippet of the kernel `producer` facilitates work distribution using the nonblocking channel write extension:

```
#pragma OPENCL EXTENSION cl_altera_channels : enable
channel long worker0, worker1;
__kernel void producer( __global const long * src )
{
    for(int i=0; i < N; i++)
    {

        bool success = FALSE;
        do
        {
            success = write_channel_nb_altera(worker0, src[i]);
            if(!success)
            {
                success = write_channel_nb_altera(worker1, src[i]);
            }
        }
        while(!success);
    }
}
```

Implementing Blocking Channel Read Extensions

The `read_channel_altera` application programming interface (API) call allows you to receive data across a channel.

Note: The read channel calls support single-call sites only. For a given channel, only one read channel call to it can exist in the entire kernel program.

- To implement a blocking channel read, include the following `read_channel_altera` function signature:
  `<type> read_channel_altera(channel <type> channel_id);`

  Where:

  `channel_id` identifies the buffer to which the channel connects, and it must match the `channel_id` of the corresponding write channel (`write_channel_altera`).

  <*type*> defines a channel data width, which cannot be a constant. Ensure that the variable the kernel assigns to read the channel data is convertible from <*type*>.

The following code snippet demonstrates the implementation of the `read_channel_altera` API call:

```
//Enables the channel extension.
#pragma OPENCL EXTENSION cl_altera_channels : enable;

//Defines chan, the kernel file-scope channel variable.
channel long chan;

/*Defines the kernel, which reads eight bytes (size of long) from the
```

```
channel and writes it back to global memory.*/
__kernel void kernel_read_channel( __global long * dst );
{
    for(int i=0; i < N; i++)
    {
        //Reads the eight bytes from the channel.
        dst[i] = read_channel_altera(chan);
    }
}
```

Caution: If the channel is empty (that is, if the FIFO buffer is empty), you cannot receive data across a read channel using the `read_channel_altera` API call. Doing so causes your kernel to stall.

Implementing Nonblocking Channel Read Extensions

Perform nonblocking reads to facilitate applications where data is not always available. The nonblocking reads signature is similar to blocking reads. However, it returns an integer value that indicates whether a read operation takes place successfully.

- To implement a blocking channel write, include the following `read_channel_nb_altera` function signature:
  `<`*`type`*`> read_channel_nb_altera(channel <`*`type`*`> channel_id, bool * valid);`

The following code snippet demonstrates the use of the nonblocking channel read extension:

```
#pragma OPENCL EXTENSION cl_altera_channels : enable
channel long chan;

__kernel void kernel_read_channel( __global long * dst )
{
    int i=0;
    while(i < N)
    {
        bool valid0, valid1;
        long data0 = read_channel_nb_altera(chan, &valid0);
        long data1 = read_channel_nb_altera(chan, &valid1);
        if (valid0)
        {
            process(data0);
        }
        if (valid1) process(data1);
        {
            process(data1);
        }
    }
}
```

Implementing I/O Channels Using the io Channels Attribute

Include an `io` attribute in your channel declaration to declare a special I/O channel to interface with input or output features of an FPGA board.
These features might include network interfaces, PCI Express (PCIe), cameras, or other data capture or processing devices or protocols.

The `io("`*`chan_id`*`")` attribute specifies the I/O feature of an accelerator board with which a channel interfaces, where *chan_id* is the name of the I/O interface listed in the **board_spec.xml** file of your Custom Platform.

Because peripheral interface usage might differ for each device type, consult your board vendor's documentation when you implement I/O channels in your kernel program. Your OpenCL kernel code must be compatible with the type of data generated by the peripheral interfaces.

Caution:
- Implicit data dependencies might exist for channels that connect to the board directly and communicate with peripheral devices via I/O channels. These implicit data dependencies might lead to compilation issues because the Altera Offline Compiler (AOC) cannot identify these dependencies.
- External I/O channels communicating with the same peripherals do not obey any sequential ordering. Ensure that the external device does not require sequential ordering because unexpected behavior might occur.

1. Consult the **board_spec.xml** file in your Custom Platform to identify the input and output features available on your FPGA board.

   For example, a **board_spec.xml** might include the following information on I/O features:

```
<channels>
  <interface name="udp_0" port="udp0_out"  type="streamsource" width="256"
   chan_id="eth0_in"/>
  <interface name="udp_0" port="udp0_in"  type="streamsink" width="256"
   chan_id="eth0_out"/>
  <interface name="udp_0" port="udp1_out"  type="streamsource" width="256"
   chan_id="eth1_in"/>
  <interface name="udp_0" port="udp1_in"  type="streamsink" width="256"
   chan_id="eth1_out"/>
</channels>
```

   The `width` attribute of an `interface` element specifies the width, in bits, of the data type used by that channel. For the example above, both the `uint` and `float` data types are 32 bits wide. Other bigger or vectorized data types must match the appropriate bit width specified in the **board_spec.xml** file.
2. Implement the `io` channel attribute as demonstrated in the following code example. The `io` channel attribute names must match those of the I/O channels (`chan_id`) specified in the **board_spec.xml** file.

```
channel QUDPWord udp_in_IO __attribute__((depth(0)))
                           __attribute__((io("eth0_in")));
channel QUDPWord udp_out_IO __attribute__((depth(0)))
                            __attribute__((io("eth0_out")));

__kernel void io_in_kernel( __global ulong4 *mem_read,
                            uchar read_from,
                            int size )
{
  int index = 0;
  ulong4 data;
  int half_size = size >> 1;
  while (index < half_size)
  {
    if (read_from & 0x01)
    {
      data = read_channel_altera(udp_in_IO);
    }
    else
    {
      data = mem_read[index];
    }
    write_channel_altera(udp_in, data);
    index++;
  }
}

__kernel void io_out_kernel( __global ulong2 *mem_write,
                                     uchar write_to,
                                     int size )
{
  int index = 0;
  ulong4 data;
  int half_size = size >> 1;
  while (index < half_size)
  {
    ulong4 data = read_channel_altera(udp_out);
    if (write_to & 0x01)
    {
      write_channel_altera(udp_out_IO, data);
    }
    else
    {
      //only write data portion
      ulong2 udp_data;
      udp_data.s0 = data.s0;
      udp_data.s1 = data.s1;
      mem_write[index] = udp_data;
    }
    index++;
  }
}
```

Attention: Declare a unique `io("`*`chan_id`*`")` handle for each I/O channel specified in the channels eXtensible Markup Language (XML) element within the **board_spec.xml** file.

Implementing Buffered Channels Using the depth Channels Attribute

You may have buffered or unbuffered channels in your kernel program. If there are imbalances in channel read and write operations, create buffered channels to prevent kernel stalls by including the `depth` attribute in your channel declaration. Buffered channels decouple the operation of concurrent work-items executing in different kernels.

You may use a buffered channel to control data traffic, such as limiting throughput or synchronizing accesses to shared memory. In an unbuffered channel, a write operation cannot proceed until the read operation reads a data value. In a buffered channel, a write operation cannot proceed until the data value is copied to the buffer. If the buffer is full, the operation cannot proceed until the read operation reads a piece of data and removes it from the channel.

- If you expect any temporary mismatch between the consumption rate and the production rate to the channel, set the buffer size using the `depth` channel attribute.
  The following example demonstrates the use of the `depth` channel attribute in kernel code that implements the Altera SDK for OpenCL (AOCL) channels extension. The `depth(N)` attribute specifies the minimum depth of a buffered channel, where *N* is the number of data values.

```
#pragma OPENCL EXTENSION cl_altera_channels : enable
channel int c __attribute__((depth(10)));

__kernel void producer( __global int * in_data )
{
    for(int i=0; i < N; i++)
    {
        if(in_data[i])
        {
            write_channel_altera(c, in_data[i]);
        }
    }
}

__kernel void consumer( __global int * restrict check_data,
                        __global int * restrict out_data )
{
    int last_val = 0;

    for(int i=0; i< N, i++)
    {
        if(check_data[i])
        {
            last_val = read_channel_altera(c);
        }
        out_data[i] = last_val;
    }
}
```

  In this example, the write operation can write ten data values to the channel without blocking. Once the channel is full, the write operation cannot proceed until an associated read operation to the channel occurs.

  Because the channel read and write calls are conditional statements, the channel might experience an imbalance between read and write calls. You may add a buffer capacity to the channel to ensure that the `producer` and `consumer` kernels are decoupled. This step is particularly important if the `producer` kernel is writing data to the channel when the `consumer` kernel is not reading from it.

Enforcing the Order of Channel Calls

To enforce the order of channel calls, introduce memory fence or barrier functions in your kernel program to control memory accesses. A memory fence function is necessary to create a control flow dependence between the channel synchronization calls before and after the fence.

When the Altera Offline Compiler (AOC) generates a compute unit, it does not create instruction-level parallelism on all instructions that are independent of each other. As a result, channel read and write operations might not execute independently of each other even if there is no control or data dependence between them. When channel calls interact with each other, or when channels write data to external devices, deadlocks might occur.

For example, the code snippet below consists of a `producer` kernel and a `consumer` kernel. Channels `c0` and `c1` are unbuffered channels. The schedule of the channel read operations from `c0` and `c1` might occur in the reversed order as the channel write operations to `c0` and `c1`. That is, the `producer` kernel writes to `c0` but the `consumer` kernel might read from `c1` first. This rescheduling of channel calls might cause a deadlock because the `consumer` kernel is reading from an empty channel.

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

__kernel void consumer( __global uint * dst,
                        const uint iterations )
{
    for(int i=0; i < iterations; i++)
    {
        /*During compilation, the AOC might reorder the way the consumer kernel
        writes to memory to optimize memory access. Therefore, c1 might be read
        before c0, which is the reverse of what appears in code.*/

        dst[2*i+1] = read_channel_altera(c0);
        dst[2*i] = read_channel_altera(c1);
    }
}
```

- To prevent deadlocks from occurring by enforcing the order of channel calls, include memory fence functions (`mem_fence`) in your kernel.
  In the kernel code above, by inserting the `mem_fence` call with the channel flag, you force the sequential ordering of the write and read channel calls in the `producer` and `consumer` kernels:

  ```
  #pragma OPENCL EXTENSION cl_altera_channels : enable

  channel uint c0 __attribute__((depth(0)));
  channel uint c1 __attribute__((depth(0)));

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

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

  In this example, `mem_fence` in the `producer` kernel ensures that the channel write operation to `c0` occurs before that to `c1`. Similarly, `mem_fence` in the `consumer` kernel ensures that the channel read operation from `c0` occurs before that from `c1`.

Defining Memory Consistency Across Kernels When Using Channels

According to the OpenCL Specification version 1.0, memory behavior is undefined unless a kernel completes execution. A kernel must finish executing before other kernels can visualize any changes in memory behavior. However, kernels that use channels can share data through common global memory buffers and synchronized memory accesses. To ensure that data written to a channel is visible to the read channel after execution passes a memory fence, define memory consistency across kernels with respect to memory fences.

- To create a control flow dependency between the channel synchronization calls and the memory operations, add the `CLK_GLOBAL_MEM_FENCE` flag to the `mem_fence` call.
  For example:

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

  In this kernel, the `mem_fence` function ensures that the write operation to `c0` and memory access to `src[2*i]` occur before the write operation to `c1` and memory access to `src[2*i+1]`. This allows data written to `c0` to be visible to the read channel before data is written to `c1`.

Use Models of AOCL Channels Implementation

Concurrent execution can improve the effectiveness of channels implementation in your OpenCL kernels. During concurrent execution, the host launches the kernels in parallel. The kernels share memory and can communicate with each other through channels where applicable.

The use models provide an overview on how to exploit concurrent execution safely and efficiently.

Feed-Forward Design Model

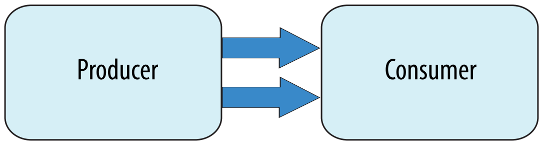
Implement the feed-forward design model to send data from one kernel to the next without creating any cycles between them. Consider the following code example:

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

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

The `producer` kernel writes data to channels `c0` and `c1`. The `consumer` kernel reads data from `c0` and `c1`. The figure below illustrates the feed-forward data flow between the two kernels:

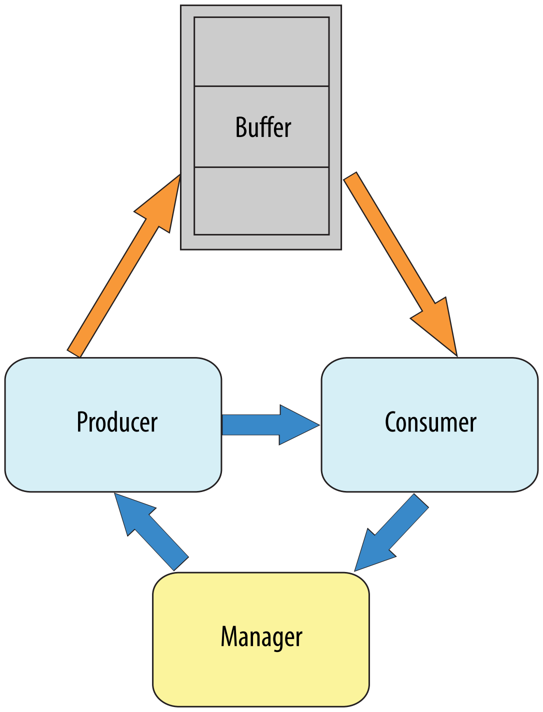
Figure 1-6: Feed-Forward Data Flow



Buffer Management

In the feed-forward design model, data traverses between the `producer` and `consumer` kernels one word at a time. To facilitate the transfer of large data messages consisting of several words, you can implement a ping-pong buffer, which is a common design pattern found in applications for communication. The figure below illustrates the interactions between kernels and a ping-pong buffer:

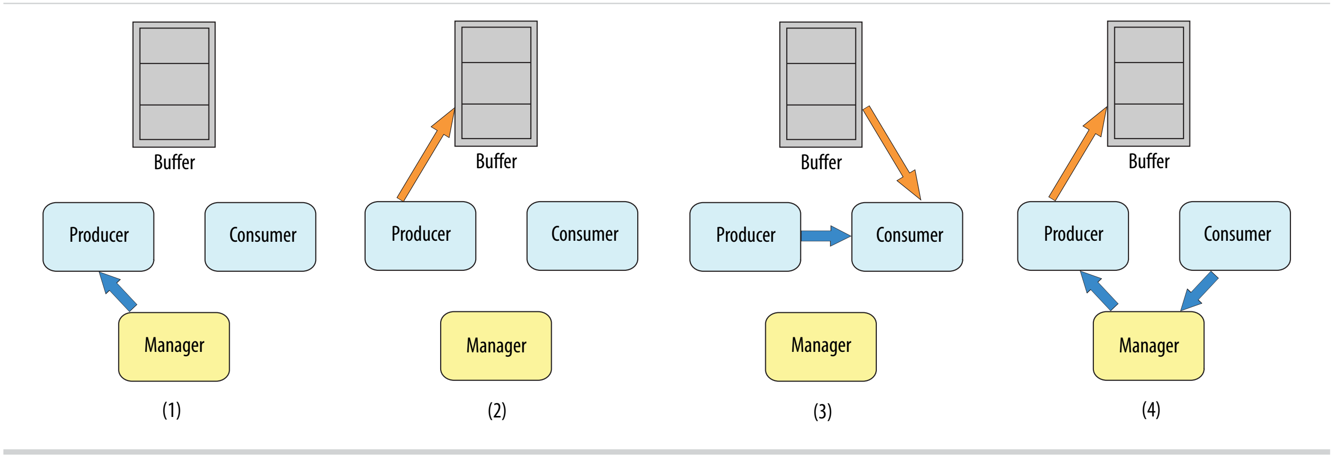
Figure 1-7: Feed-Forward Design Model with Buffer Management



The `manager` kernel manages circular buffer allocation and deallocation between the `producer` and `consumer` kernels. After the `consumer` kernel processes data, the `manager` receives memory regions that the `consumer` frees up and sends them to the `producer` for reuse. The `manager` also sends to the `producer` kernel the initial set of free locations, or tokens, to which the `producer` can write data.

The following figure illustrates the sequence of events that take place during buffer management:

Figure 1-8: Kernels Interaction during Buffer Management



1. The `manager` kernel sends a set of tokens to the `producer` kernel to indicate initially which regions in memory are free for `producer` to use.
2. After `manager` allocates the memory region, `producer` writes data to that region of the ping-pong buffer.
3. After `producer` completes the write operation, it sends a synchronization token to the `consumer` kernel to indicate what memory region contains data for processing. The `consumer` kernel then reads data from that region of the ping-pong buffer.

   Note: When `consumer` is performing the read operation, `producer` can write to other free memory locations for processing because of the concurrent execution of the `producer`, `consumer`, and `manager` kernels.
4. After `consumer` completes the read operation, it releases the memory region and sends a token back to the `manager` kernel. The `manager` kernel then recycles that region for `producer` to use.

Implementation of Buffer Management for AOCL Kernels

To ensure that the Altera SDK for OpenCL (AOCL) implements buffer management properly, the ordering of channel read and write operations is important. The synchronization token must occur after the `producer` kernel commits data to memory. In other words, the channel write operation cannot occur until `producer` stores its data successfully. To preserve this ordering, include an OpenCL `mem_fence` token in your kernels, as shown below:

```
__kernel void producer( __global const uint * restrict src,
                        __global volatile uint * restrict shared_mem,
                        const uint iterations )
{
    int base_offset;

    for (uint gID = 0; gID < iterations; gID++)
    {
        // Assume each block of memory is 256 words
        uint lID = 0x0ff & gID;

        if(lID == 0)
        {
            base_offset = read_channel_altera(req);
        }
```

```
            shared_mem[base_offset + lID] = src[gID];

            // Make sure all memory operations are committed before
            // sending token to the consumer
            mem_fence(CLK_GLOBAL_MEM_FENCE | CLK_CHANNEL_MEM_FENCE);

            if (lID == 255)
            {
                write_channel_altera(c, base_offset);
            }
        }
    }
```

The `mem_fence` construct takes two flags: `CLK_GLOBAL_MEM_FENCE` and `CLK_CHANNEL_MEM_FENCE`. The `mem_fence` effectively creates a control flow dependence between operations that occur before and after the `mem_fence` call. The `CLK_GLOBAL_MEM_FENCE` flag indicates that global memory operations must obey the control flow. The `CLK_CHANNEL_MEM_FENCE` indicates that channel operations must obey the control flow. As a result, the `write_channel_altera` call in the example cannot start until the global memory operation is committed to the shared memory buffer.

Implementing OpenCL Pipes

The Altera SDK for OpenCL (AOCL) provides preliminary support for OpenCL pipe functions. OpenCL pipes are part of the OpenCL Specification version 2.0. They provide a mechanism for passing data to kernels and synchronizing kernels with high efficiency and low latency.

Implement pipes if it is important that your OpenCL kernel is compatible with other SDKs.

Refer to the *OpenCL Specification version 2.0* for OpenCL C programming language specification and general information about pipes.

The AOCL implementation of pipes does not encompass the entire pipes specification. As such, it is not fully conformant to the OpenCL Specification version 2.0. The goal of the AOCL pipes implementation is to provide a solution that works seamlessly on a different OpenCL 2.0-conformant device. To enable pipes for Altera devices, your design must satisfy certain additional requirements.

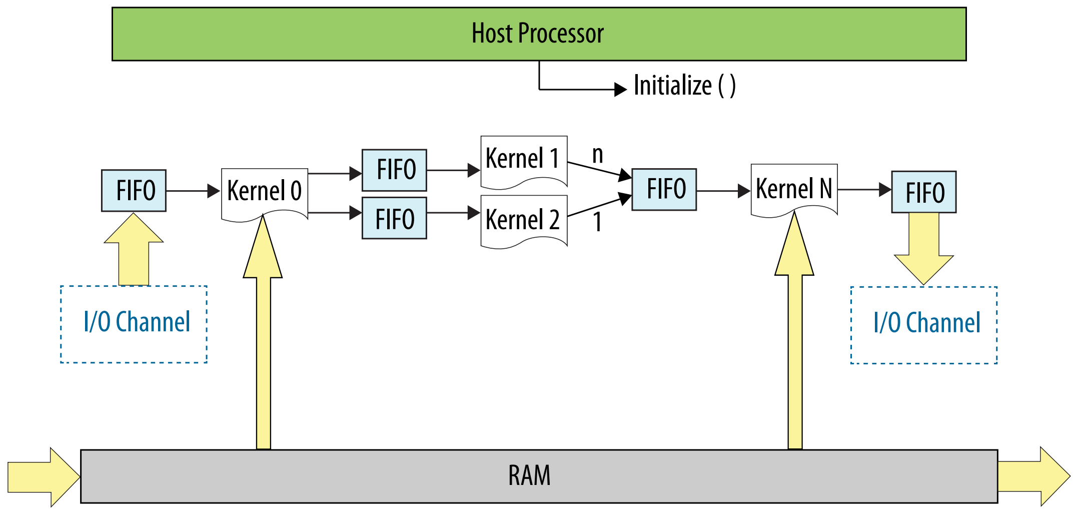
Related Information

OpenCL Specification version 2.0 (API)

Overview of the OpenCL Pipe Functions

OpenCL pipes allow kernels to communicate directly with each other via FIFO buffers.

Figure 1-9: Pipes Network Example



Implementation of pipes decouples kernel execution from the host processor. The foundation of the Altera SDK for OpenCL (AOCL) pipes support is the AOCL channels extension. However, the syntax for pipe functions differs from the channels syntax.

Important: Unlike channels, pipes have a default nonblocking behavior.

For more information on blocking and nonblocking functions, refer to the corresponding documentation on channels.

Related Information

- **Implementing Blocking Channel Write Extensions** on page 1-23
- **Implementing Nonblocking Channel Write Extensions** on page 1-24
- **Implementing Nonblocking Channel Read Extensions** on page 1-26
- **Implementing Blocking Channel Read Extensions** on page 1-25

Pipe Data Behavior

Data written to a pipe remains in a pipe as long as the kernel program remains loaded on the FPGA device. In other words, data written to a pipe persists across multiple work-groups and NDRange invocations. However, data is not persistent across multiple or different invocations of kernel programs.

Consider the following code example:

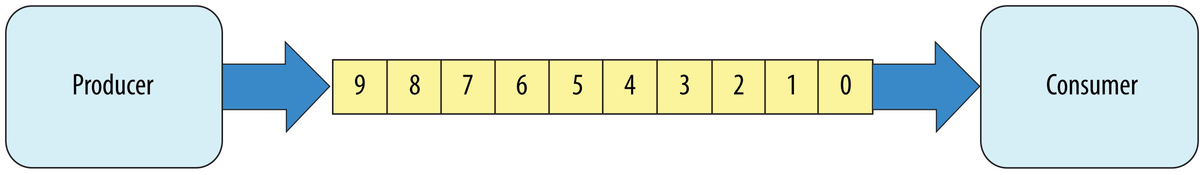
```
__kernel void
producer (write_only pipe uint __attribute__((blocking)) c0)
{
    for (uint i=0;i<10;i++)
    {
        write_pipe( c0, &i );
    }
}

__kernel void
consumer (__global uint * restrict dst,
          read_only pipe uint __attribute__((blocking))
```

```
                                      __attribute__((depth(10))) c0)
{
    for (int i=0;i<5;i++)
    {
        read_pipe( c0, &dst[i] );
    }
}
```

A read operation to a pipe reads the *least* recent piece of data written to the pipe first. Pipes data maintains their FIFO ordering within the pipe. The figure below illustrates the order in which the `producer` kernel writes the elements to the pipe:

Figure 1-10: Pipe Data FIFO Ordering



The kernel `producer` writes ten elements ([0, 9]). The kernel `consumer` reads five elements from the pipe per NDRange invocation. During the first invocation, the kernel `consumer` reads values 0 to 4 from the pipe. Because the data persists across NDRange invocations, the second time you execute the kernel `consumer`, it reads values 5 to 9.

For this example, to avoid a deadlock from occurring, you need to invoke the kernel `consumer` twice for every invocation of the kernel `producer`. If you call `consumer` less than twice, `producer` stalls because the pipe becomes full. If you call `consumer` more than twice, `consumer` stalls because there is insufficient data in the pipe.

Multiple Work-Item Ordering for Pipes

The OpenCL specification does not define a work-item ordering. The Altera SDK for OpenCL (AOCL) enforces a work-item order to maintain the consistency in pipe read and write operations.

Multiple work-item accesses to a pipe can be useful in some scenarios. For example, they are useful when data words in the pipe are independent, or when the pipe is implemented for control logic. The main concern regarding multiple work-item accesses to a pipe is the order in which the kernel writes data to and reads data from the pipe. If possible, the OpenCL pipes process work-items read and write operations to a pipe in a deterministic order. As such, the read and write operations remain consistent across kernel invocations.

Requirements for Deterministic Multiple Work-Item Ordering

To guarantee deterministic ordering, the AOCL checks that the pipe call is work-item invariant based on the following characteristics:

1. Work-items must pass through a pipe call before exiting the kernel function.
2. Work-items must pass through a pipe call before entering the ensuing kernel function.
3. If either of the preceding characteristics is not satisfied, the AOCL checks that all branch conditions to the pipe call basic block is work-item invariant.

If the AOCL cannot guarantee deterministic ordering of multiple work-item accesses to a pipe, it warns you that the pipes might not have well-defined ordering with nondeterministic execution. Primarily, the

AOCL fails to provide deterministic ordering if you have work-item-variant code on loop executions with pipe calls, as illustrated below:

```
__kernel void
ordering (__global int * check, global int * data,
          write_only pipe int __attribute__((blocking)) req)
{
    int condition = check[get_global_id(0)];

    if (condition)
    {
        for (int i=0;i<N;i++)
        {
            process(data);
            write_pipe( req, &data[i] );
        }
    }
    else
    {
        process(data);
    }
}
```

Because the Altera Offline Compiler (AOC) performs many transformations, such as branch conversion, during kernel invocations, it might be difficult to determine if the requirements are fulfilled for a given pipe call. The AOCL generates a graphical report on pipe connectivity across multiple kernels.

Work-Item Serial Execution of Pipes

Work-item serial execution refers to an ordered execution behavior where work-item sequential IDs determine their execution order in the compute unit.

When you implement pipes in a kernel, the Altera Offline Compiler (AOC) enforces that kernel behavior is equivalent to having at most one work-group in flight. The AOC also ensures that the kernel executes pipes in work-item serial execution, where the kernel executes work-items with smaller IDs first. A work-item has the identifier `(x, y, z, group)`, where `(x, y, z)` is the local 3D identifier, and `group` is the work-group identifier.

The work-item ID `(x0, y0, z0, group0)` is considered to be smaller than the ID `(x1, y1, z1, group1)` if one of the following conditions is true:

- `group0` < `group1`
- `group0` = `group1` and `z0` < `z1`
- `group0` = `group1` and `z0` = `z1` and `y0` < `y1`
- `group0` = `group1` and `z0` = `z1` and `y0` = `y1` and `x0` < `x1`

For example, the work-item with an ID `(x0, y0, z0, group0)` executes the write pipe call first, and then the work-item with an ID `(x1, y0, z0, group0)` executes the call, and so on, in a sequential order. Defining this order ensures that the system is verifiable with external models.

Pipe Execution in Loop with Multiple Work-Items

When pipes exist in the body of a loop with multiple work-items, as shown below, each loop iteration executes prior to subsequent iterations. This implies that loop iteration 0 of each work-item in a work-group executes before iteration 1 of each work-item in a work-group, and so on.

```
__kernel void
ordering (__global int * data,
          write_only pipe int __attribute__((blocking)) req)
```

```
{
    write_pipe( req, &data[get_global_id(0)] );
}
```

Restrictions in OpenCL Pipes Implementation

There are certain design restrictions to the implementation of pipes in your OpenCL application.

Default Behavior

By default, pipes exhibit nonblocking behavior. If you want the pipes in your kernel to exhibit blocking behavior, specify the blocking attribute (`__attribute__((blocking))`) when you declare the read and write pipes.

Emulation Support

Currently, the Altera SDK for OpenCL (AOCL) Emulator does not support emulation of kernels that contain pipes.

Pipes API Support

Currently, the AOCL implementation of pipes does not support all the built-in pipe functions in the OpenCL Specification version 2.0. For a list of supported and unsupported pipe APIs, refer to *OpenCL Programming Language Restrictions for Pipes*.

Single Call Site

Because the pipe read and write operations do not function deterministically, for a given kernel, you can only assign one call site per pipe ID. For example, the Altera Offline Compiler (AOC) cannot compile the following code example:

```
in_data1 = read_pipe(pipe1);
in_data2 = read_pipe(pipe2);
in_data3 = read_pipe(pipe1);
```

The second `read_pipe` call to `pipe1` causes compilation failure because it creates a second call site to `pipe1`.

To gather multiple data from a given pipe, divide the pipe into multiple pipes, as shown below:

```
in_data1 = read_pipe(pipe1);
in_data2 = read_pipe(pipe2);
in_data3 = read_pipe(pipe3);
```

Because you can only assign a single call site per pipe ID, you cannot unroll loops containing pipes. Consider the following code:

```
#pragma unroll 4
for (int i=0; i < 4; i++)
{
    in_data = read_pipe(pipe1);
}
```

The AOC issues the following warning message during compilation:

```
Compiler Warning: Unroll is required but the loop cannot be unrolled.
```

Feedback and Feed-Forward Channels

Pipes within a kernel can be either `read_only` or `write_only`. Performance of a kernel that reads and writes to the same pipe is poor.

Kernel Vectorization Support

You cannot vectorize kernels that use pipes; that is, do not include the `num_simd_work_items` kernel attribute in your kernel code. Vectorizing a kernel that uses pipes creates multiple pipe masters and requires arbitration, which OpenCL pipes specification does not support.

Instruction-Level Parallelism on read_pipe and write_pipe Calls

If no data dependencies exist between `read_pipe` and `write_pipe` calls, the AOC attempts to execute these instructions in parallel. As a result, the AOC might execute these `read_pipe` and `write_pipe` calls in an order that does not follow the sequence expressed in the OpenCL kernel code.

Consider the following code sequence:

```
in_data1 = read_pipe(pipe1);
in_data2 = read_pipe(pipe2);
in_data3 = read_pipe(pipe3);
```

Because there are no data dependencies between the `read_pipe` calls, the AOC can execute them in any order.

Related Information

OpenCL C Programming Language Restrictions for Pipes on page 2-6

Enabling OpenCL Pipes for Kernels

To implement pipes, modify your OpenCL kernels to include pipes-specific application programming interface (API) calls.

Pipes declarations are unique within a given OpenCL kernel program. Also, pipe instances are unique for every OpenCL kernel program-device pair. If the runtime loads a single OpenCL kernel program onto multiple devices, each device will have a single copy of each pipe. However, these pipe copies are independent and do not share data across the devices.

Ensuring Compatibility with Other OpenCL SDKs

Altera's implementation of OpenCL pipes is a beta feature. As such, the implementation is currently partially conformant to the OpenCL Specification version 2.0. If you port a kernel that implements pipes from another OpenCL SDK to the Altera SDK for OpenCL (AOCL), you must modify the host code and the kernel code. The modifications do not affect subsequent portability of your application to other OpenCL SDKs.

Host Code Modification

Below is an example of a modified host application:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "CL/opencl.h"
#define SIZE 1000

const char *kernel_source = "__kernel void pipe_writer(__global int *in,"
                            "                           write_only pipe int p_in)\n"
                            "{\n"
                            "    int gid = get_global_id(0);\n"
                            "    write_pipe(p_in, &in[gid]);\n"
                            "}\n"
                            "__kernel void pipe_reader(__global int *out,"
                            "                           read_only pipe int p_out)\n"
                            "{\n"
                            "    int gid = get_global_id(0);\n"
                            "    read_pipe(p_out, &out[gid]);\n"
                            "}\n";

int main()
{
    int *input = (int *)malloc(sizeof(int) * SIZE);
    int *output = (int *)malloc(sizeof(int) * SIZE);
    memset(output, 0, sizeof(int) * SIZE);
    for (int i = 0; i != SIZE; ++i)
    {
        input[i] = rand();
    }

    cl_int status;
    cl_platform_id platform;
    cl_uint num_platforms;
    status = clGetPlatformIDs(1, &platform, &num_platforms);

    cl_device_id device;
    cl_uint num_devices;
    status = clGetDeviceIDs(platform,
                            CL_DEVICE_TYPE_ALL,
                            1,
                            &device,
                            &num_devices);

    cl_context context = clCreateContext(0, 1, &device, NULL, NULL, &status);

    cl_command_queue queue = clCreateCommandQueue(context, device, 0, &status);

    size_t len = strlen(kernel_source);
    cl_program program = clCreateProgramWithSource(context,
                                                   1,
                                                   (const char **)&kernel_source,
                                                   &len,
                                                   &status);

    status = clBuildProgram(program, num_devices, &device, "", NULL, NULL);

    cl_kernel pipe_writer = clCreateKernel(program, "pipe_writer", &status);
    cl_kernel pipe_reader = clCreateKernel(program, "pipe_reader", &status);

    cl_mem in_buffer = clCreateBuffer(context,
                                      CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                                      sizeof(int) * SIZE,
                                      input,
                                      &status);
    cl_mem out_buffer = clCreateBuffer(context,
                                       CL_MEM_WRITE_ONLY,
                                       sizeof(int) * SIZE,
                                       NULL,
                                       &status);

    cl_mem pipe = clCreatePipe(context, 0, sizeof(cl_int), SIZE, NULL, &status);

    status = clSetKernelArg(pipe_writer, 0, sizeof(cl_mem), &in_buffer);
    status = clSetKernelArg(pipe_writer, 1, sizeof(cl_mem), &pipe);
    status = clSetKernelArg(pipe_reader, 0, sizeof(cl_mem), &out_buffer);
    status = clSetKernelArg(pipe_reader, 1, sizeof(cl_mem), &pipe);

    size_t size = SIZE;
    cl_event sync;
    status = clEnqueueNDRangeKernel(queue,
                                    pipe_writer,
                                    1,
                                    NULL,
                                    &size,
                                    &size,
                                    0,
                                    NULL,
                                    &sync);
    status = clEnqueueNDRangeKernel(queue,
                                    pipe_reader,
                                    1,
                                    NULL,
                                    &size,
                                    &size,
                                    1,
                                    &sync,
                                    NULL);
    status = clFinish(queue);

    status = clEnqueueReadBuffer(queue,
                                 out_buffer,
                                 CL_TRUE,
                                 0,
                                 sizeof(int) * SIZE,
                                 output,
                                 0,
                                 NULL,
                                 NULL);

    int golden = 0, result = 0;
    for (int i = 0; i != SIZE; ++i)
    {
      golden += input[i];
      result += output[i];
    }

    int ret = 0;
    if (golden != result)
    {
        printf("FAILED!");
        ret = 1;
    } else
    {
        printf("PASSED!");
    }
    printf("\n");

    return ret;
}
```

Kernel Code Modification

The kernel code runs on OpenCL SDKs that conforms to the OpenCL Specification version 2.0. Before running this kernel on the AOCL, perform the following modifications:

- Rename the pipe arguments so that they are the same in both kernels. For example, rename `p_in` and `p_out` to `p`.
- Specify the depth attribute for the pipe arguments. Assign a depth attribute value that equals to the maximum number of packets that the pipe creates to hold in the host.
- Execute the kernel program in the offline compilation mode because the AOCL has an offline compiler.

The modified kernel code appears as follows:

```
#define SIZE 1000

__kernel void pipe_writer(__global int *in,
                          write_only pipe int __attribute__((depth(SIZE))) p)
{
    int gid = get_global_id(0);
    write_pipe(p, &in[gid]);
}

__kernel void pipe_reader(__global int *out,
                          read_only pipe int __attribute__((depth(SIZE))) p)
{
    int gid = get_global_id(0);
    read_pipe(p, &out[gid]);
}
```

Declaring the Pipe Handle

Use the `pipe` variable to define the static pipe connectivity between kernels or between kernels and I/O.

To read from and write to a pipe, the kernel must pass the pipe variable to each of the corresponding application programming interface (API) call.

- Declare the pipe handle as a file scope variable in the kernel source code in the following convention: `<access qualifier> pipe <type> <variable_name>`

  The *<type>* of the pipe may be any OpenCL built-in scalar or vector data type with a scalar size of 1024 bits or less. It may also be any user-defined type that is comprised of scalar or vector data type with a scalar size of 1024 bits or less.

  Consider the following pipe handle declarations:

  ```
  __kernel void first (pipe int c)
  ```

  ```
  __kernel void second (write_only pipe int c)
  ```

  The first example declares a read-only pipe handle of type `int` in the kernel `first`. The second example declares a write-only pipe in the kernel `second`. The kernel `first` may only read from pipe `c`, and the kernel `second` may only write to pipe `c`.

  Important: The Altera Offline Compiler (AOC) statically infers the connectivity of pipes in your system by matching the names of the pipe arguments. In the example above, the kernel `first` is connected to the kernel `second` by the pipe `c`.

In an Altera OpenCL system, only one kernel may read to a pipe. Similarly, only one kernel may write to a pipe. If a non-IO pipe does not have at least one corresponding reading operation and one writing operation, the AOC issues an error.

Implementing Pipe Writes

The `write_pipe` application programming interface (API) call allows you to send data across a pipe.

Altera only supports the convenience version of the `write_pipe` function. By default, `write_pipe` calls are nonblocking. Pipe write operations are successful only if there is capacity in the pipe to hold the incoming packet.

Attention: The write pipe calls support single-call sites only. For a given pipe, only one write pipe call to it can exist in the entire kernel program.

- To implement a pipe write, include the following `write_pipe` function signature:

  `int write_pipe (write_only pipe <`*`type`*`> pipe_id, const <`*`type`*`> *data);`

  Where:

  `pipe_id` identifies the buffer to which the pipe connects, and it must match the `pipe_id` of the corresponding read pipe (`read_pipe`).

  `data` is the data that the pipe write operation writes to the pipe. It is a pointer to the packet type of the pipe. Note that writing to the pipe might lead to a global or local memory load, depending on the source address space of the data pointer.

  <*type*> defines a pipe data width. The return value indicates whether the pipe write operation is successful. If successful, the return value is 0. If pipe write is unsuccessful, the return value is -1.

The following code snippet demonstrates the implementation of the `write_pipe` API call:

```
/*Declares the writable nonblocking pipe, p, which contains packets of type
int*/
__kernel void kernel_write_pipe (__global const long *src,
                                 write_only pipe int p)
{
    for (int i=0; i < N; i++)
    {
        //Performs the actual writing
        //Emulates blocking behavior via the use of a while loop
        while (write_pipe(p, &src[i]) < 0) { }
    }
}
```

The `while` loop is unnecessary if you specify a *blocking attribute*. To facilitate better hardware implementations, Altera provides facility for blocking `write_pipe` calls by specifying the blocking attribute (that is, `__attribute__((blocking))`) on the pipe arugment declaration for the kernel. Blocking `write_pipe` calls always return success.

Caution: When you send data across a blocking write pipe using the `write_pipe` API call, keep in mind that if the pipe is full (that is, if the FIFO buffer is full of data), your kernel will stall. Use the Altera SDK for OpenCL (AOCL) Profiler to check for pipe stalls.

Related Information

Profiling Your OpenCL Kernel on page 1-89

Implementing Pipe Reads

The `read_pipe` application programming interface (API) call allows you to receive data across a channel.

Altera only supports the convenience version of the `read_pipe` function. By default, `read_pipe` calls are nonblocking.

Note: The read pipe calls support single-call sites only. For a given pipe, only one read pipe call to it can exist in the entire kernel program.

- To implement a pipe read, include the following `read_pipe` function signature:

  `int read_pipe (read_only_pipe <`*type*`> pipe_id, <`*type*`> *data);`

  Where:

  `pipe_id` identifies the buffer to which the pipe connects, and it must match the `pipe_id` of the corresponding pipe write operation (`write_pipe`).

  `data` is the data that the pipe read operation reads from the pipe. It is a pointer to location of the data. Note that `write_pipe` call might lead to a global or local memory load, depending on the source address space of the data pointer.

  <*type*> defines the packet size of the data.

The following code snippet demonstrates the implementation of the `read_pipe` API call:

```
/*Declares the read_only_pipe that contains packets
of type long.*/
/*Declares that read_pipe calls within the kernel will exhibit
blocking behavior*/
__kernel void kernel_read_pipe (__global long *dst,
                                read_only pipe long
__attribute__((blocking)) p)
{
    for (int i=0; i < N; i++)
    {
        /*Reads from a long from the pipe and stores it
        into global memory at the specified location*/
        read_pipe(p, &dst[i]);
    }
}
```

To facilitate better hardware implementations, Altera provides facility for blocking `write_pipe` calls by specifying the blocking attribute (that is, `__attribute__((blocking))`) on the pipe arugment declaration for the kernel. Blocking `write_pipe` calls always return success.

Caution: If the pipe is empty (that is, if the FIFO buffer is empty), you cannot receive data across a blocking read pipe using the `read_pipe` API call. Doing so causes your kernel to stall.

Implementing Buffered Pipes Using the depth Attribute

You may have buffered or unbuffered pipes in your kernel program. If there are imbalances in pipe read and write operations, create buffered pipes to prevent kernel stalls by including the `depth` attribute in your pipe declaration. Buffered pipes decouple the operation of concurrent work-items executing in different kernels.

You may use a buffered pipe to control data traffic, such as limiting throughput or synchronizing accesses to shared memory. In an unbuffered pipe, a write operation can only proceed when the read operation is expecting to read data. Use unbuffered pipes in conjunction with blocking read and write behaviors in kernels that execute concurrently. The unbuffered pipes provide self-synchronizing data transfers efficiently.

In a buffered pipe, a write operation can only proceed if there is capacity in the pipe to hold the incoming packet. A read operation can only proceed if there is at least one packet in the pipe. Use buffered pipes if pipe calls are predicated differently in the writer and reader kernels, and the kernels do not execute concurrently.

- If you expect any temporary mismatch between the consumption rate and the production rate to the pipe, set the buffer size using the `depth` attribute.
  The following example demonstrates the use of the `depth` attribute in kernel code that implements the OpenCL pipes. The `depth(`*`N`*`)` attribute specifies the minimum depth of a buffered channel, where *N* is the number of data values. If the read and write kernels specify different depths for a given buffered pipe, the Altera Offline Compiler (AOC) will use the larger depth of the two.

```
__kernel void
producer (__global int *in_data,
          write_only pipe int __attribute__((blocking))
                              __attribute__((depth(10))) c)
{
    for (i=0; i < N; i++)
    {
        if (in_data[i])
        {
            write_pipe( c, &in_data[i] );
        }
    }
}

__kernel void
consumer (__global int *check_data,
          __global int *out_data,
          read_only pipe int __attribute__((blocking)) c )
{
    int last_val = 0;
    for (i=0; i < N; i++)
    {
        if (check_data[i])
        {
            read_pipe( c, &last_val );
        }
        out_data[i] = last_val;
    }
}
```

  In this example, the write operation can write ten data values to the pipe successfully. After the pipe is full, the write kernel returns failure until a read kernel consumes some of the data in the pipe.

  Because the pipe read and write calls are conditional statements, the pipe might experience an imbalance between read and write calls. You may add a buffer capacity to the pipe to ensure that the `producer` and `consumer` kernels are decoupled. This step is particularly important if the `producer` kernel is writing data to the pipe when the `consumer` kernel is not reading from it.

Implementing I/O Pipes Using the io Attribute

Include an `io` attribute in your OpenCL pipe declaration to declare a special I/O pipe to interface with input or output features of an FPGA board.
These features might include network interfaces, PCI Express (PCIe), cameras, or other data capture or processing devices or protocols.

In the Altera SDK for OpenCL channels extension, the `io("`*`chan_id`*`")` attribute specifies the I/O feature of an accelerator board with which a channel interfaces, where *chan_id* is the name of the I/O interface listed in the **board_spec.xml** file of your Custom Platform. The same I/O features can be used to identify I/O pipes.

Because peripheral interface usage might differ for each device type, consult your board vendor's documentation when you implement I/O pipes in your kernel program. Your OpenCL kernel code must be compatible with the type of data generated by the peripheral interfaces. If there is a difference in the byte ordering between the external I/O channels and the kernel, the Altera Offline Compiler (AOC) converts the byte ordering seamlessly upon entry and exit.

Caution:
- Implicit data dependencies might exist for pipes that connect to the board directly and communicate with peripheral devices via I/O pipes. These implicit data dependencies might lead to compilation issues because the Altera Offline Compiler (AOC) cannot identify these dependencies.
- External I/O channels communicating with the same peripherals do not obey any sequential ordering. Ensure that the external device does not require sequential ordering because unexpected behavior might occur.

1. Consult the **board_spec.xml** file in your Custom Platform to identify the input and output features available on your FPGA board.

   For example, a **board_spec.xml** might include the following information on I/O features:

   ```
   <channels>
     <interface name="udp_0" port="udp0_out"  type="streamsource" width="256"
      chan_id="eth0_in"/>
     <interface name="udp_0" port="udp0_in"  type="streamsink" width="256"
      chan_id="eth0_out"/>
     <interface name="udp_0" port="udp1_out"  type="streamsource" width="256"
      chan_id="eth1_in"/>
     <interface name="udp_0" port="udp1_in"  type="streamsink" width="256"
      chan_id="eth1_out"/>
   </channels>
   ```

   The `width` attribute of an `interface` element specifies the width, in bits, of the data type used by that pipe. For the example above, both the `uint` and `float` data types are 32 bits wide. Other bigger or vectorized data types must match the appropriate bit width specified in the **board_spec.xml** file.
2. Implement the `io` attribute as demonstrated in the following code example. The `io` attribute names must match those of the I/O channels (`chan_id`) specified in the **board_spec.xml** file.

   ```
   __kernel void test (pipe uint pkt   __attribute__((io(“enet”))),;
                       pipe float data __attribute__((io(“pcie”))));
   ```

   Attention: Declare a unique `io("`*`chan_id`*`")` handle for each I/O pipe specified in the channels eXtensible Markup Language (XML) element within the **board_spec.xml** file.

Enforcing the Order of Pipe Calls

To enforce the order of pipe calls, introduce memory fence or barrier functions in your kernel program to control memory accesses. A memory fence function is necessary to create a control flow dependence between the pipe synchronization calls before and after the fence.

When the Altera Offline Compiler (AOC) generates a compute unit, it does not create instruction-level parallelism on all instructions that are independent of each other. As a result, pipe read and write operations might not execute independently of each other even if there is no control or data dependence between them. When pipe calls interact with each other, or when channels write data to external devices, deadlocks might occur.

For example, the code snippet below consists of a `producer` kernel and a `consumer` kernel. Pipes `c0` and `c1` are unbuffered pipes. The schedule of the pipe read operations from `c0` and `c1` might occur in the reversed order as the pipe write operations to `c0` and `c1`. That is, the `producer` kernel writes to `c0` but the `consumer` kernel might read from `c1` first. This rescheduling of pipe calls might cause a deadlock because the `consumer` kernel is reading from an empty pipe.

```
__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 < iterations; 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+1] );
        read_pipe( c1, &dst[2*i] );
    }
}
```

- To prevent deadlocks from occurring by enforcing the order of pipe calls, include memory fence functions (`mem_fence`) in your kernel.
  In the kernel code above, by inserting the `mem_fence` call with the pipe flag, you force the sequential ordering of the write and read pipe calls in the `producer` and `consumer` kernels:

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

__kernel void
consumer (__global uint * 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  ]);
        mem_fence(CLK_CHANNEL_MEM_FENCE);
        read_pipe(c1, &dst[2*i+1]);
    }
}
```

  In this example, `mem_fence` in the `producer` kernel ensures that the pipe write operation to `c0` occurs before that to `c1`. Similarly, `mem_fence` in the `consumer` kernel ensures that the pipe read operation from `c0` occurs before that from `c1`.

Defining Memory Consistency Across Kernels When Using Pipes

According to the OpenCL Specification version 2.0, memory behavior is undefined unless a kernel completes execution. A kernel must finish executing before other kernels can visualize any changes in memory behavior. However, kernels that use pipes can share data through common global memory buffers and synchronized memory accesses. To ensure that data written to a pipe is visible to the read pipe after execution passes a memory fence, define memory consistency across kernels with respect to memory fences.

- To create a control flow dependency between the pipe synchronization calls and the memory operations, add the `CLK_GLOBAL_MEM_FENCE` flag to the `mem_fence` call.
  For example:

```
__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<iterations;i++)
    {
        write_pipe( c0, &src[2*i  ] );
        mem_fence( CLK_CHANNEL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE );
        write_pipe( c1, &src[2*i+1] );
    }
}
```

  In this kernel, the `mem_fence` function ensures that the write operation to `c0` and memory access to `src[2*i]` occur before the write operation to `c1` and memory access to `src[2*i+1]`. This allows data written to `c0` to be visible to the read pipe before data is written to `c1`.

Using Predefined Preprocessor Macros in Conditional Compilation

You may take advantage of predefined preprocessor macros that allow you to conditionally compile portions of your kernel code.

- To include device-specific (for example, FPGA_board_1) code in your kernel program, structure your kernel program in the following manner:

```
#if defined(AOCL_BOARD_FPGA_board_1)
    //FPGA_board_1-specific statements
#else
    //FPGA_board_2-specific statements
#endif
```

  When you target your kernel compilation to a specific board, it sets the predefined preprocessor macro `AOCL_BOARD_<board_name>` to 1. If <*board_name*> is FPGA_board_1, the Altera Offline Compiler (AOC) will compile the FPGA_board_1-specific parameters and features.
- To introduce AOC-specific compiler features and optimizations, structure your kernel program in the following manner:

```
#if defined(ALTERA_CL)
    //statements
#else
    //statements
#endif
```

  Where `ALTERA_CL` is the Altera predefined preprocessor macro for the AOC.

Related Information

Defining Preprocessor Macros to Specify Kernel Parameters (-D <macro_name>) on page 1-79

Declaring __constant Address Space Qualifiers

There are several limitations and workarounds you must consider when you include `__constant` address space qualifiers in your kernel.

Function Scope __constant Variables

The Altera Offline Compiler (AOC) does not support function scope `__constant` variables. Replace function scope `__constant` variables with file scope constant variables. You can also replace function scope `__constant` variables with `__constant` buffers that the host passes to the kernel.

File Scope __constant Variables

If the host always passes the same constant data to your kernel, consider declaring that data as a constant preinitialized file scope array within the kernel file. Declaration of a constant preinitialized file scope array creates a ROM directly in the hardware to store the data. This ROM is available to all work-items in the NDRange.

The AOC supports only scalar file scope constant data. For example, you may set the `__constant` address space qualifier as follows:

```
__constant int my_array[8] = {0x0, 0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7};

__kernel void my_kernel (__global int * my_buffer)
{
   size_t gid = get_global_id(0);
   my_buffer[gid] += my_array[gid % 8];
}
```

In this case, the AOC sets the values for `my_array` in a ROM because the file scope constant data does not change between kernel invocations.

Warning: Do not set your file scope `__constant` variables in the following manner because the AOC does not support vector type `__constant` arrays declared at the file scope:

```
__constant int2 my_array[4] = {(0x0, 0x1), (0x2, 0x3); (0x4, 0x5), (0x6,
0x7)};
```

Pointers to __constant Parameters from the Host

You can replace file scope constant data with a pointer to a `__constant` parameter in your kernel code. You must then modify your host application in the following manner:

1. Create `cl_mem` memory objects associated with the pointers in global memory.
2. Load constant data into `cl_mem` objects with `clEnqueueWriteBuffer` prior to kernel execution.
3. Pass the `cl_mem` objects to the kernel as arguments with the `clSetKernelArg` function.

For simplicity, if a constant variable is of a complex type, use a `typedef` argument, as shown in the table below:

Table 1-1: Replacing File Scope __constant Variable with Pointer to __constant Parameter

If your source code is structured as follows:	Rewrite your code to resemble the following syntax:

`__constant int Payoff[2][2] = {{ 1, 3}, {5, 3}};`
`__kernel void original(__global int * A)`
`{`
`   *A = Payoff[1][2];`
`   // and so on`
`}`
	`__kernel void modified(__global int * A,`
	`__constant Payoff_type * PayoffPtr )`
	`{`
	`   *A = (PayoffPtr)[1][2];`
	`   // and so on`
	`}`

Attention: Use the same type definition in both your host application and your kernel.

Including Structure Data Types as Arguments in OpenCL Kernels

Convert each structure parameter (`struct`) to a pointer that points to a structure.

The table below describes how you can convert structure parameters:

Table 1-2: Converting Structure Parameters to Pointers that Point to Structures

If your source code is structured as follows:

```
struct Context
{
   float param1;
   float param2;
   int param3;
   uint param4;
};

__kernel void algorithm(__global float
* A,
struct Context c)
{
   if ( c.param3 )
   {
      // statements
   }
}
```

Rewrite your code to resemble the following syntax:

```
struct Context
{
   float param1;
   float param2;
   int param3;
   uint param4;
};

__kernel void algorithm(__global float
* A,
__global struct Context * restrict c)
{
   if ( c->param3 )
   {
      // Dereference through a
      // pointer and so on
   }
}
```

Attention: The `__global struct` declaration creates a new buffer to store the structure. To prevent pointer aliasing, include a `restrict` qualifier in the declaration of the pointer to the structure.

Matching Data Layouts of Host and Kernel Structure Data Types

If you use structure data types (`struct`) as arguments in OpenCL kernels, match the member data types and align the data members between the host application and the kernel code.

To match member data types, use the `cl_` version of the data type in your host application that corresponds to the data type in the kernel code. The `cl_` version of the data type is available in the **opencl.h** header file. For example, if you have a data member of type `float4` in your kernel code, the corresponding data member you declare in the host application is `cl_float4`.

Align the structures and align the `struct` data members between the host and kernel applications. Manage the alignments carefully because of the variability among different host compilers.

For example, if you have `float 4` OpenCL data types in the struct, the alignments of these data items must satisfy the OpenCL specification (that is, 16-byte alignment for `float4`).

The following rules apply when the Altera Offline Compiler (AOC) compiles your OpenCL kernels:

1. Alignment of built-in scalar and vector types follow the rules outlined in Section 6.1.5 of the *OpenCL Specification version 1.0*.

   The AOC usually aligns a data type based on its size. However, the AOC aligns a value of a three-element vector the same way it aligns a four-element vector.
2. An array has the same alignment as one of its elements.
3. A `struct` (or a `union`) has the same alignment as the maximum alignment necessary for any of its data members.

Consider the following example:

```
struct my_struct
{
    char data[3];
    float4 f4;
    int index;
};
```

The AOC aligns the `struct` elements above at 16-byte boundaries because of the `float4` data type. As a result, both `data` and `index` also have 16-byte alignment boundaries.

4. The AOC does not reorder data members of a `struct`.
5. Normally, the AOC inserts a minimum amount of data structure padding between data members of a `struct` to satisfy the alignment requirements for each data member.

    a. In your OpenCL kernel code, you may specify data packing (that is, no insertion of data structure padding) by applying the `packed` attribute to the `struct` declaration. If you impose data packing, ensure that the alignment of data members satisfies the OpenCL alignment requirements. The Altera SDK for OpenCL (AOCL) does not enforce these alignment requirements. Ensure that your host compiler respects the kernel attribute and sets the appropriate alignments.

    b. In your OpenCL kernel code, you may specify the amount of data structure padding by applying the `aligned(N)` attribute to a data member, where *N* is the amount of padding. The AOCL does not enforce these alignment requirements. Ensure that your host compiler respects the kernel attribute and sets the appropriate alignments.

    For Windows systems, some versions of the Microsoft Visual Studio compiler pack structure data types by default. If you do not want to apply data packing, specify an amount of data structure padding as shown below:

```
struct my_struct
{
    __declspec(align(16)) char data[3];

    /*Note that cl_float4 is the only known float4 definition on the host*/
    __declspec(align(16)) cl_float4 f4;

    __declspec(align(16)) int index;
};
```

Tip: An alternative way of adding data structure padding is to insert dummy `struct` members of type `char` or array of `char`.

Related Information

Modifying Host Program for Structure Parameter Conversion on page 1-62

Disabling Insertion of Data Structure Padding

You may instruct the Altera Offline Compiler (AOC) to disable automatic padding insertion between members of a `struct` data structure.

- To disable automatic padding insertion, insert the `packed` attribute prior to the kernel source code for a `struct` data structure.
  For example:

```
__attribute__((packed))
struct Context
{
   float param1;
   float param2;
   int param3;
   uint param4;
};
__kernel void algorithm(__global float * restrict A, __global struct Context *
restrict c)
{
   if ( c->param3 )
   {
      // Dereference through a pointer and so on
   }
}
```

For more information, refer to the *Align a Struct with or without Padding* section of the *Altera SDK for OpenCL Best Practices Guide.*

Related Information

Align a Struct with or without Padding

Specifying the Alignment of a Struct

You may instruct the Altera Offline Compiler (AOC) to set a specific alignment of a `struct` data structure.

- To specify the struct alignment, insert the `aligned(N)` attribute prior to the kernel source code for a `struct` data structure.
  For example:

```
__attribute__((aligned(2)))
struct Context
{
   float param1;
   float param2;
   int param3;
   uint param4;
};
__kernel void algorithm(__global float * A, _global struct Context * restrict c)
{
   if ( c->param3 )
   {
      // Dereference through a pointer and so on
   }
}
```

For more information, refer to the *Align a Struct with or without Padding* section of the *Altera SDK for OpenCL Best Practices Guide.*

Related Information

Align a Struct with or without Padding

Inferring a Register

The Altera Offline Compiler (AOC) can implement data that is in the private address space in registers or in block RAMs. In general, the AOC chooses registers if the access to a variable is fixed and does not require any dynamic indexes. Accessing an array with a variable index usually forces the array into block RAMs. Implementing private data as registers is beneficial for data access that occurs in a single cycle (for example, feedback in a single work-item loop).

The AOC infers private arrays as registers either as single values or in a piecewise fashion. Piecewise implementation results in very efficient hardware; however, the AOC must be able to determine data accesses statically. To facilitate piecewise implementation, hardcode the access points into the array. You can also facilitate register inference by unrolling loops that access the array.

If array accesses are not inferable statically, the AOC might infer the array as registers. However, the AOC limits the size of these arrays to 64 bytes in length for single work-item kernels. There is effectively no size limit for kernels with multiple work-items

Consider the following code example:

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

The indexing into `array[i]` is not inferable statically because the loop is not unrolled. If the size of `array[i]` is less than or equal to 64 bytes for single work-item kernels, the AOC implements `array[i]` in block RAMs. If the size of `array[i]` is greater than 64 bytes, or if the kernel has multiple work-items, the AOC implements the entire array into registers as a single value. In this case, the AOC implements data accesses as nonconstant shifts and masks. With complicated addressing, the AOC implements the array in block RAMs and instantiates specialized hardware for each load or store operation.

Inferring a Shift Register

The shift register design pattern is a very important design pattern for many applications. However, the implementation of a shift register design pattern might seem counterintuitive at first.

Consider the following code example:

```
channel int in, out;

#define SIZE 512
//Shift register size must be statically determinable

__kernel void foo()
{
    int shift_reg[SIZE];
       //The key is that the array size is a compile time constant

    // Initialization loop
    #pragma unroll
    for (int i=0; i < SIZE; i++)
    {
        //All elements of the array should be initialized to the same value
        shift_reg[i] = 0;
    }
    while(1)
    {
        // Fully unrolling the shifting loop produces constant accesses
        #pragma unroll
        for (int j=0; j < SIZE–1; j++)
        {
            shift_reg[j] = shift_reg[j + 1];
        }
        shift_reg[SIZE – 1] = read_channel_altera(in);

        // Using fixed access points of the shift register
        int res = (shift_reg[0] + shift_reg[1]) / 2;

        // ‘out’ channel will have running average of the input channel
        write_channel_altera(out, res);
    }
}
```

In each clock cycle, the kernel shifts a new value into the array. By placing this shift register into a block RAM, the Altera Offline Compiler (AOC) can efficiently handle multiple access points into the array. The shift register design pattern is ideal for implementing filters (for example, image filters like a Sobel filter or time-delay filters like a finite impulse response (FIR) filter).

When implementing a shift register in your kernel code, keep in mind the following key points:

1. Unroll the shifting loop so that it can access every element of the array.
2. All access points must have constant data accesses. For example, if you write a calculation in nested loops using multiple access points, unroll these loops to establish the constant access points.
3. Initialize all elements of the array to the same value. Alternatively, you may leave the elements uninitialized if you do not require a specific initial value.
4. If some accesses to a large array are not inferable statically, they force the AOC to create inefficient hardware. If these accesses are necessary, use `__local` memory instead of `__private` memory.
5. Do not shift a large shift register conditionally. The shifting must occur in very loop iteration that contains the shifting code to avoid creating inefficient hardware.

Enabling Double Precision Floating-Point Operations

The Altera SDK for OpenCL offers preliminary support for all double precision floating-point functions.

Before declaring any double precision floating-point data type in your OpenCL kernel, include the following `OPENCL EXTENSION` pragma in your kernel code:

```
#pragma OPENCL EXTENSION cl_khr_fp64 : enable
```

Designing Your Host Application

Altera offers guidelines on host requirements and procedures on structuring the host application. If applicable, implement these design strategies when you create or modify a host application for your OpenCL kernels.

Host Programming Requirements on page 1-57
When designing your OpenCL host application for use with the Altera SDK for OpenCL (AOCL), ensure that the application satisfies the following host programming requirements.

Allocating OpenCL Buffer for Manual Partitioning of Global Memory on page 1-58

Creating a Pipe Object in Your Host Application on page 1-60
To implement OpenCL pipes in your kernel, you must create Altera SDK for OpenCL (AOCL)-specific pipe objects in your host application.

Collecting Profile Data During Kernel Execution on page 1-60
In cases where kernel execution finishes after the host application completes, you can query the FPGA explicitly to collect profile data during kernel execution.

Accessing Custom Platform-Specific Functions on page 1-62
To reference Custom Platform-specific user-accessible functions while linking to the ACD, include the `clGetBoardExtensionFunctionAddressAltera` extension in your host application.

Modifying Host Program for Structure Parameter Conversion on page 1-62
If you convert any structure parameters to pointers-to-constant structures in your OpenCL kernel, you must modify your host application accordingly.

Allocating Shared Memory for OpenCL Kernels Targeting SoCs on page 1-63
Altera recommends that OpenCL kernels that run on Altera SoCs access shared memory instead of the FPGA DDR memory.

Managing Host Application on page 1-65
The Altera SDK for OpenCL (AOCL) includes utility commands you can invoke to obtain information on flags and libraries necessary for compiling and linking your host application.

Host Programming Requirements

When designing your OpenCL host application for use with the Altera SDK for OpenCL (AOCL), ensure that the application satisfies the following host programming requirements.

Host Machine Memory Requirements

The machine that runs the host application must have enough host memory to support several components simultaneously.

The host machine must support the following components:

- The host application and operating system.
- The working set for the host application.
- The maximum amount of OpenCL memory buffers that can be allocated at once. Every device-side `cl_mem` buffer is associated with a corresponding storage area in the host process. Therefore, the amount of host memory necessary might be as large as the amount of external memory supported by the FPGA.

Host Binary Requirement

When compiling the host application, target one of these architectures: x86-64 (64-bit), big-endian (64-bit), or ARM® 32-bit ARMV7-A for devices such as the Cyclone V SoC. The Altera SDK for OpenCL (AOCL) host runtime does not support x86-32 (32-bit) binaries.

Multiple Host Threads

The Altera SDK for OpenCL (AOCL) host library is not thread-safe.

If you have a multi-threaded host application, Altera recommends that you build your own external synchronization mechanism around all OpenCL host function calls.

Out-of-Order Command Queues

The OpenCL host runtime command queues do not support out-of-order command execution.

Requirement for Multiple Command Queues in Channels or Pipes Implementation

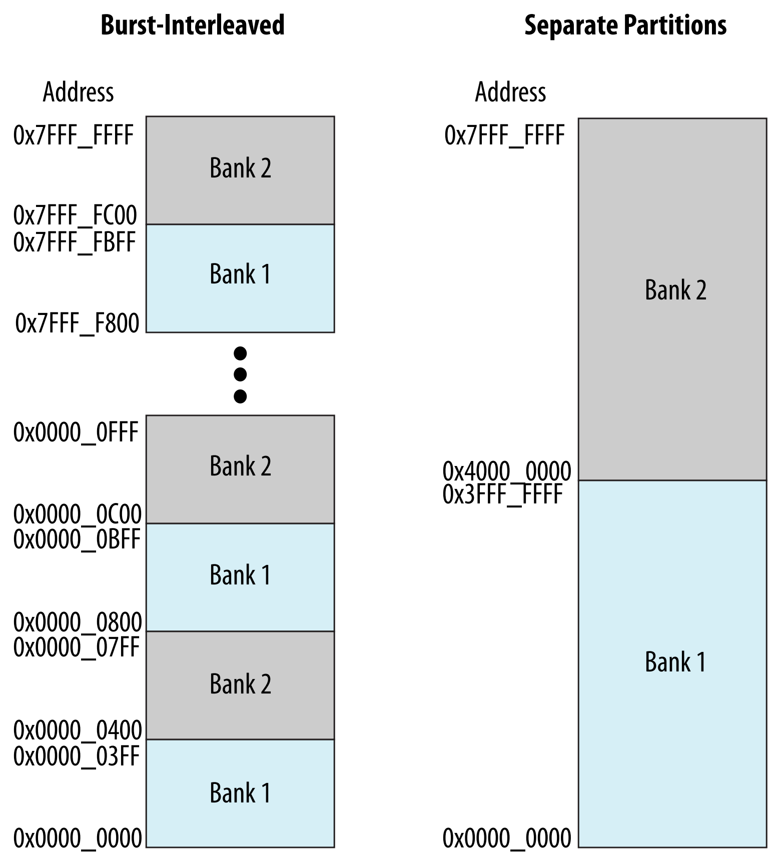
Although the Altera SDK for OpenCL (AOCL) channels extension or OpenCL pipes implementation allows multiple kernels to execute in parallel, channels or pipes facilitate this concurrent behavior only when `cl_command_queue` objects are in order. To enable multiple command queues , instantiate a separate command for each kernel you wish to run concurrently.

Allocating OpenCL Buffer for Manual Partitioning of Global Memory

Manual partitioning of global memory buffers allows you to control memory accesses across buffers to maximize the memory bandwidth. Before you partition the memory, first you have to disable burst-interleaving during OpenCL kernel compilation. Then, in the host application, you must specify the memory bank to which you allocate the OpenCL buffer.

By default, the Altera Offline Compiler (AOC) configures each global memory type in a burst-interleaved fashion. Usually, the burst-interleaving configuration leads to the best load balancing between the memory banks. However, there might be situations where it is more efficient to partition the memory into non-interleaved regions.

The figure below illustrates the differences between burst-interleaved and non-interleaved memory partitions.



To manually partition some or all of the available global memory types, perform the following tasks:

1. Compile your OpenCL kernel using the `--no-interleaving <global_memory_type>` flag to configure the memory bank(s) of the specified memory type as separate addresses.

   For more information on the usage of the `--no-interleaving <global_memory_type>` flag, refer to the *Disabling Burst-Interleaving of Global Memory (--no-interleaving <global_memory_type>)* section.
2. Create an OpenCL buffer in your host application, and allocate the buffer to one of the banks using the `CL_MEM_HETEROGENEOUS_ALTERA` and `CL_MEM_BANK` flags.
   - Specify `CL_MEM_BANK_1_ALTERA` to allocate the buffer to the lowest available memory region.
   - Specify `CL_MEM_BANK_2_ALTERA` to allocation memory to the second bank (if available).

   Attention: Allocate each buffer to a single memory bank only.

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

```
mem = clCreateBuffer(context,
                     flags|CL_MEM_HETEROGENEOUS_ALTERA,
                     memSize,
                     NULL,
                     &errNum);
```

   For example, the following `clCreateBuffer` call allocates memory into the lowest available memory region of a nondefault memory bank:

```
mem = clCreateBuffer(context,
           (CL_MEM_HETEROGENEOUS_ALTERA|CL_MEM_BANK_1_ALTERA),
           memSize,
           NULL,
           &errNum);
```

   The `clCreateBuffer` call allocates memory into a certain global memory type based on what you specify in the kernel argument. If a memory (`cl_mem`) object residing in a memory type is set as a kernel argument that corresponds to a different memory technology, the host moves the memory object automatically when it queues the kernel. Do not pass a buffer as kernel arguments that associate it with multiple memory technologies.

Attention: If the second bank is not available at runtime, the memory is allocated to the first bank. If no global memory is available, the `clCreateBuffer` call fails with the error message `CL_MEM_OBJECT_ALLOCATION_FAILURE`.

For more information on optimizing heterogeneous global memory accesses, refer to the *Heterogeneous Memory Buffers* and the *Manual Partitioning of Global Memory* sections of the *Altera SDK for OpenCL Best Practices Guide*.

Related Information

- **Disabling Burst-Interleaving of Global Memory (--no-interleaving <global_memory_type>)** on page 1-82

- Manual Partitioning of Global Memory
- Heterogeneous Memory Buffers

Creating a Pipe Object in Your Host Application

To implement OpenCL pipes in your kernel, you must create Altera SDK for OpenCL (AOCL)-specific pipe objects in your host application.

An AOCL-specific pipe object is not a true OpenCL pipe object as described in the OpenCL Specification version 2.0. This implementation allows you to migrate away from Altera devices with a conformant solution. The AOCL-specific pipe object is a memory object (`cl_mem`); however, the host does not allocate any memory for the pipe itself.

The following `clCreatePipe` host application programming interface (API) creates a pipe object:

```
cl_mem clCreatePipe(cl_context context,
                    cl_mem_flags flags,
                    cl_uint pipe_packet_size,
                    cl_uint pipe_max_packets,
                    const cl_pipe_properties *properties,
                    cl_int *errcode_ret)
```

For more information on the `clCreatePipe` host API function, refer to section 5.4.1 of the *OpenCL Specification version 2.0*.

Below is an example syntax of the clCreatePipe host API function:

```
cl_int status;
cl_mem c0_pipe = clCreatePipe(context,
                              0,
                              sizeof(int),
                              1,
                              NULL,
                              &status);
status = clSetKernelArg(kernel, 1, sizeof(cl_mem), &c0_pipe);
```

Caution: The AOCL does not support dynamic channel assignment at runtime. The AOCL statically links the pipes during compilation.

Related Information

OpenCL Specification version 2.0 (API)

Collecting Profile Data During Kernel Execution

In cases where kernel execution finishes after the host application completes, you can query the FPGA explicitly to collect profile data during kernel execution.

When you profile your OpenCL kernel during compilation, a **profile.mon** file is generated automatically. The profile data is then written to **profile.mon** after kernel execution completes on the FPGA. However, if kernel execution completes after the host application completes, no profiling information for that kernel invocation will be available in the **profile.mon** file. In this case, you can modify your host code to acquire profiling information during kernel execution.

- To query the FPGA to collect profile data while the kernel is running, call the following host library call:

```
extern CL_API_ENTRY cl_int CL_API_CALL

clGetProfileInfoAltera(cl_event);
```

where `cl_event` is the kernel event. The kernel event you pass to this host library call must be the same one you pass to the `clEnqueueNDRangeKernel` call.

Important: If kernel execution completes before the invocation of `clGetProfileInfoAltera`, the function returns an event error message.

Caution: Invoking the `clGetProfileInfoAltera` function during kernel execution disables the profile counters momentarily so that the Profiler can collect data from the FPGA. As a result, you will lose some profiling information during this interruption. If you call this function at very short intervals, the profile data might not accurately reflect the actual performance behavior of the kernel.

Consider the following example host code:

```
int main()
{   ...
    clEnqueueNDRangeKernel (queue, kernel, ..., NULL);
    ...
    clEnqueueNDRangeKernel (queue, kernel, .. , NULL);
    ...
}
```

This host application runs on the assumption that a kernel launches twice and then completes. In the **profile.mon** file, there will be two sets of profile data, one for each kernel invocation. To collect profile data while the kernel is running, modify the host code in the following manner:

```
int main()
{
    ...
    clEnqueueNDRangeKernel (queue, kernel, ..., &event);

    //Get the profile data before the kernel completes
    clGetProfileInfoAltera (event);

    //Wait until the kernel completes
    clFinish (queue);

    ...
    clEnqueueNDRangeKernel (queue, kernel, ..., NULL);
    ...
}
```

The call to `clGetProfileInfoAltera` adds a new entry in the **profile.mon** file. The Profiler GUI then parses this entry in the report.

For more information on the Altera SDK for OpenCL (AOCL) Profiler, refer to the following sections:

- *Profile Your Kernel to Identify Performance Bottlenecks* in the *Altera SDK for OpenCL Best Practices Guide*
- *Profiling Your OpenCL Kernel*

Related Information

- **Profile Your Kernel to Identify Performance Bottlenecks**
- **Profiling Your OpenCL Kernel** on page 1-89

Accessing Custom Platform-Specific Functions

You have the option to include in your application user-accessible functions that are available in your Custom Platform. However, when you link your host applicaiton to the Altera Client Driver (ACD), you cannot directly reference these Custom Platform-specific functions. To reference Custom Platform-specific user-accessible functions while linking to the ACD, include the `clGetBoardExtensionFunctionAddressAltera` extension in your host application.

The `clGetBoardExtensionFunctionAddressAltera` extension specifies an application programming interface (API) that retrieves a pointer to a user-accessible function from the Custom Platform.

Attention: For Linux systems, the `clGetBoardExtensionFunctionAddressAltera` function works with or without ACD. For Windows systems, the function only works in conjunction with ACD. Consult with your board vendor to determine if ACD is supported in your Custom Platform.

Definitions of the extension interfaces are available in the ***ALTERAOCLSDKROOT*/host/include/CL/cl_ext.h** file.

- To obtain a pointer to a user-accessible function in your Custom Platform, call the following function in your host application:

```
void* clGetBoardExtensionFunctionAddressAltera (
    const char* function_name,
    cl_device_id device
    );
```

  Where:

  function_name is the name of the user-accessible function that your Custom Platform vendor provides,

  and

  device is the device ID returned by the `clGetDeviceIDs` function.

  After locating the user-accessible function, the `clGetBoardExtensionFunctionAddressAltera` function returns a pointer to the user-accessible function. If the function does not exist in the Custom Platform, `clGetBoardExtensionFunctionAddressAltera` returns `NULL`.

Modifying Host Program for Structure Parameter Conversion

If you convert any structure parameters to pointers-to-constant structures in your OpenCL kernel, you must modify your host application accordingly.

Perform the following changes to your host application:

1. Allocate a `cl_mem` buffer to store the structure contents.

Attention: You need a separate `cl_mem` buffer for every kernel that uses a different structure value.

2. Set the structure kernel argument with a pointer to the structure buffer, not with a pointer to the structure contents.
3. Populate the structure buffer contents before queuing the kernel. Perform one of the following steps to ensure that the structure buffer is populated before the kernel launches:
    - Queue the structure buffer on the same command queue as the kernel queue.
    - Synchronize separate kernel queues and structure buffer queues with an event.
4. When your application no longer needs to call a kernel that uses the structure buffer, release the `cl_mem` buffer.

Related Information

- **Including Structure Data Types as Arguments in OpenCL Kernels** on page 1-52
- **Matching Data Layouts of Host and Kernel Structure Data Types** on page 1-52

Allocating Shared Memory for OpenCL Kernels Targeting SoCs

Altera recommends that OpenCL kernels that run on Altera SoCs access shared memory instead of the FPGA DDR memory. FPGA DDR memory is accessible to kernels with very high bandwidths. However, read and write operations from the ARM CPU to FPGA DDR memory are very slow because they do not use direct memory access (DMA). Reserve FPGA DDR memory only for passing temporary data between kernels or within a single kernel for testing purposes.

Before you begin

Note:
1. Mark the shared buffers between kernels as volatile to ensure that buffer modification by one kernel is visible to the other kernel.
2. To access shared memory, you only need to modify the host code. Modifications to the kernel code are unnecessary.
3. You cannot use the library function `malloc` or the operator `new` to allocate physically shared memory. Also, the `CL_MEM_USE_HOST_PTR` flag does not work with shared memory.

    In DDR memory, shared memory must be physically contiguous. The FPGA cannot consume virtually contiguous memory without a scatter-gather direct memory access (SG-DMA) controller core. The `malloc` function and the `new` operator are for accessing memory that is virtually contiguous.
4. CPU caching is disabled for the shared memory.

The ARM CPU and the FPGA can access the shared memory simultaneously. You do not need to include the `clEnqueueReadBuffer` and `clEnqueueWriteBuffer` calls in your host code to make data visible to either the FPGA or the CPU.

- To allocate and access shared memory, structure your host code in a similar manner as the following example:

  ```
  cl_mem src = clCreateBuffer(…, CL_MEM_ALLOC_HOST_PTR, size, …);
  int *src_ptr  = (int*)clEnqueueMapBuffer  (…, src, size, …);
  *src_ptr = input_value; //host writes to ptr directly
  clSetKernelArg (…, src);
  clEnqueueNDRangeKernel(…);
  clFinish();
  printf (“Result = %d\n”, *dst_ptr); //result is available immediately
  clEnqueueUnmapMemObject(…, src, src_ptr, …);
  clReleaseMemObject(src); // actually frees physical memory
  ```

  You can include the `CONFIG_CMA_SIZE_MBYTES` kernel configuration option to control the maximum total amount of shared memory available for allocation. In practice, the total amount of allocated shared memory is smaller than the value of `CONFIG_CMA_SIZE_MBYTES`.

  Important:
  1. If your target board has multiple DDR memory banks, the `clCreateBuffer(..., CL_MEM_READ_WRITE, ...)` function allocates memory to the nonshared DDR memory banks. However, if the FPGA has access to a single DDR bank that is shared memory, then `clCreateBuffer(..., CL_MEM_READ_WRITE, ...)` allocates to shared memory, similar to using the `CL_MEM_ALLOC_HOST_PTR` flag.
  2. The shared memory that you request with the `clCreateBuffer(..., CL_MEM_ALLOC_HOST_PTR, size, ...)` function is allocated in the Linux OpenCL kernel driver, and it relies on the contiguous memory allocator (CMA) feature of the Linux kernel. For detailed information on enabling and configuring the CMA, refer to the *Recompiling the Linux Kernel and the OpenCL Linux Kernel Driver* section of the *Altera Cyclone V SoC Development Kit Reference Platform Porting Guide*.

- To transfer data from shared hard processor system (HPS) DDR to FPGA DDR efficiently, include a kernel that performs the `memcpy` function, as shown below.

  ```
  __attribute__((num_simd_work_items(8)))
  mem_stream(__global uint * src, __global uint * dst)
  {
      size_t gid = get_global_id(0);
      dst[gid] = src[gid];
  }
  ```

  Attention: Allocate the `src` pointer in the HPS DDR as shared memory using the `CL_MEM_ALLOC_HOST_PTR` flag.

- If the host allocates constant memory to shared HPS DDR system and then modifies it after kernel execution, the modifications might not take effect. As a result, subsequent kernel executions might use outdated data. To prevent kernel execution from using outdated constant memory, perform one of the following tasks:

  1. Do not modify constant memory after its initialization.
  2. Create multiple constant memory buffers if you require multiple `__constant` data sets.
  3. If available, allocate constant memory to the FPGA DDR on your accelerator board.

Related Information

Recompiling the Linux Kernel and the OpenCL Linux Kernel Driver

Managing Host Application

The Altera SDK for OpenCL (AOCL) includes utility commands you can invoke to obtain information on flags and libraries necessary for compiling and linking your host application.

Attention: To cross-compile your host application to an SoC board, include the `--arm` option in your utility command.

Caution: For Linux systems, if you debug your host application using the GNU Project Debugger (GDB), invoke the following command prior to running the host application:

```
handle SIG44 nostop
```

Without this command, the GDB debugging process terminates with the following error message:

```
Program received signal SIG44, Real-time event 44.
```

Displaying Example Makefile Fragments (example-makefile or makefile)

To display example Makefile fragments for compiling and linking a host application against host runtime libraries available with the Altera SDK for OpenCL, invoke the `example-makefile` or `makefile` utility command.

- At a command prompt, invoke the `aocl example-makefile` or `aocl makefile` utility command.
  The software displays an output similar to the following:

```
The following are example Makefile fragments for compiling and linking
a host program against the host runtime libraries included with the
Altera SDK for OpenCL.


Example GNU makefile on Linux, with GCC toolchain:

   AOCL_COMPILE_CONFIG=$(shell aocl compile-config)
   AOCL_LINK_CONFIG=$(shell aocl link-config)

   host_prog : host_prog.o
        g++ -o host_prog host_prog.o $(AOCL_LINK_CONFIG)

   host_prog.o : host_prog.cpp
        g++ -c host_prog.cpp $(AOCL_COMPILE_CONFIG)


Example GNU makefile on Windows, with Microsoft Visual C++ command line compiler:

   AOCL_COMPILE_CONFIG=$(shell aocl compile-config)
   AOCL_LINK_CONFIG=$(shell aocl link-config)

   host_prog.exe : host_prog.obj
        link -nologo /OUT:host_prog.exe host_prog.obj $(AOCL_LINK_CONFIG)

   host_prog.obj : host_prog.cpp
        cl /MD /Fohost_prog.obj -c host_prog.cpp $(AOCL_COMPILE_CONFIG)


Example GNU makefile cross-compiling to ARM SoC from Linux or Windows, with
Linaro GCC cross-compiler toolchain:

   CROSS-COMPILER=arm-linux-gnueabihf-
   AOCL_COMPILE_CONFIG=$(shell aocl compile-config --arm)
   AOCL_LINK_CONFIG=$(shell aocl link-config --arm)

   host_prog : host_prog.o
        $(CROSS-COMPILER)g++ -o host_prog host_prog.o $(AOCL_LINK_CONFIG)

   host_prog.o : host_prog.cpp
        $(CROSS-COMPILER)g++ -c host_prog.cpp $(AOCL_COMPILE_CONFIG)
```

Compiling and Linking Your Host Application

The OpenCL host application uses standard OpenCL runtime application programming interfaces (APIs) to manage device configuration, data buffers, kernel launches, and synchronization. The host application also contains functions such as file I/O, or portions of the source code that do not run on an accelerator device. The Altera SDK for OpenCL (AOCL) includes utility commands you can invoke to obtain information on C header files describing the OpenCL APIs, and board-specific memory-mapped device (MMD) and host runtime libraries with which you must link your host application.

Important: For Windows systems, you must add the `/MD` flag to link the host runtime libraries against the multithreaded dynamically-linked library (DLL) version of the Microsoft C Runtime library. You must also compile your host application with the `/MD` compilation flag, or use the `/NODEFAULTLIB` linker option to override the selection of runtime library.

Remember: Include the path to the ***ALTERAOCLSDKROOT*/host/<*OS_platform*>/bin** folder in your library search path when you run your host application.

Displaying Flags for Compiling Host Application (compile-config) on page 1-67
To display a list of flags necessary for compiling a host application, invoke the `compile-config` utility command.

Displaying Paths to OpenCL Host Runtime and MMD Libraries (ldflags) on page 1-67
To display the paths necessary for linking a host application to the OpenCL host runtime and memory-mapped device (MMD) libraries, invoke the `ldflags` utility command.

Listing OpenCL Host Runtime and MMD Libraries (ldlibs) on page 1-67
To display the names of the OpenCL host runtime and memory-mapped device (MMD) libraries necessary for linking a host application, invoke the `ldlibs` utility command.

Displaying Information on OpenCL Host Runtime and MMD Libraries (link-config or linkflags) on page 1-68
To display a list of flags necessary for linking a host application with OpenCL host runtime and memory-mapped device (MMD) libraries, invoke the `link-config` or `linkflags` utility command.

Displaying Flags for Compiling Host Application (compile-config)

To display a list of flags necessary for compiling a host application, invoke the `compile-config` utility command.

1. At a command prompt, invoke the `aocl compile-config` utility command.
   The software displays the path to the folder or directory in which the OpenCL application programming interface (API) header files reside. For example:
   - For Windows systems, the path is `-I%`*`ALTERAOCLSDKROOT`*`%/host/include`
   - For Linux systems, the path is `-I$`*`ALTERAOCLSDKROOT`*`/host/include`

   where *ALTERAOCLSDKROOT* points to the location of the software installation.
2. Add this path to your C preprocessor.

Attention: In your host source, include the **opencl.h** OpenCL header file, located in the ***ALTERAOCLSDKROOT*/host/include/CL** folder or directory.

Displaying Paths to OpenCL Host Runtime and MMD Libraries (ldflags)

To display the paths necessary for linking a host application to the OpenCL host runtime and memory-mapped device (MMD) libraries, invoke the `ldflags` utility command.

- At a command prompt, invoke the `aocl ldflags` utility command.
  The software displays the paths for linking your host application with the following libraries:
  1. The OpenCL host runtime libraries that provide OpenCL platform and runtime application programming interfaces (APIs). The OpenCL host runtime libraries are available in the ***ALTERAOCLSDKROOT*/host/<*OS_platform*>/lib** directory.
  2. The path to the Custom Platform-specific MMD libraries. The MMD libraries are available in the **<*board_family_name*>/<*OS_platform*>/lib** directory of your Custom Platform.

Listing OpenCL Host Runtime and MMD Libraries (ldlibs)

To display the names of the OpenCL host runtime and memory-mapped device (MMD) libraries necessary for linking a host application, invoke the `ldlibs` utility command.

- At a command prompt, invoke the `aocl ldlibs` utility command.
  The software lists the OpenCL host runtime libraries residing in the ***ALTERAOCLSDKROOT*/host/*<OS_platform>*/lib** directory. It also lists the Custom Platform-specific MMD libraries residing in the **/*<board_family_name>*/*<OS_platform>*/lib** directory of your Custom Platform.

  - For Windows systems, the output might resemble the following example:

    ```
    alterahalmmd.lib
    <board_vendor_name>_<board_family_name>_mmd.[lib|so|a|dll]
    alteracl.lib
    acl_emulator_kernel_rt.lib
    pkg_editor.lib
    libelf.lib
    acl_hostxml.lib
    ```

  - For Linux systems, the output might resemble the following example:

    ```
    -lalteracl
    -ldl
    -lacl_emulator_kernel_rt
    -lalterahalmmd
    -l<board_vendor_name>_<board_family_name>_mmd
    -lelf
    -lrt
    -lstdc++
    ```

Displaying Information on OpenCL Host Runtime and MMD Libraries (link-config or linkflags)

To display a list of flags necessary for linking a host application with OpenCL host runtime and memory-mapped device (MMD) libraries, invoke the `link-config` or `linkflags` utility command.

This utility command combines the functions of the `ldflags` and `ldlibs` utility commands.

1. At a command prompt, invoke the `aocl link-config` or `aocl linkflags` command.
   The software displays the link options for linking your host application with the following libraries:

   1. The path to and the names of OpenCL host runtime libraries that provide OpenCL platform and runtime application programming interfaces (APIs). The OpenCL host runtime libraries are available in the ***ALTERAOCLSDKROOT*/host/*<OS_platform>*/lib** directory .
   2. The path to and the names of the Custom Platform-specific MMD libraries. The MMD libraries are available in the ***<board_family_name>*/*<OS_platform>*/lib** directory of your Custom Platform.

- For Windows systems, the link options might resemble the following example output:

```
/libpath:%ALTERAOCLSDKROOT%/board/<board_name>/windows64/lib
/libpath:%ALTERAOCLSDKROOT%/host/windows64/lib
alterahalmmd.lib
<board_vendor_name>_<board_family_name>_mmd.[lib|so|a|dll]
alteracl.lib
acl_emulator_kernel_rt.lib
pkg_editor.lib
libelf.lib
acl_hostxml.lib
```

- For Linux systems, the link options might resemble the following example output:

```
-L/$ALTERAOCLSDKROOT/board/<board_name>/linux64/lib
-L/$ALTERAOCLSDKROOT/host/linux64/lib
-lalterac
-ldl
-lacl_emulator_kernel_rt
-lalterahalmmd
-l<board_vendor_name>_<board_family_name>_mmd
-lelf
-lrt
-lstdc++
```

Programming an FPGA via the Host

The Altera Offline Compiler (AOC) is an offline compiler that compiles kernels independently of the host application. To load the kernels into the OpenCL runtime, include the `clCreateProgramWithBinary` function in your host application.

Caution: If your host system consists of multiple processors, only one processor can access the FPGA at a given time. Consider an example where there are two host applications, corresponding to two processors, attempting to launch kernels onto the same FPGA at the same time. The second host application wil receive an error message indicating that the device is busy. The second host application cannot run until the first host application releases the OpenCL context.

1. Compile your OpenCL kernel with the AOC to create the Altera Offline Compiler Executable file (**.aocx**).
2. Include the `clCreateProgramWithBinary` function in your host application to create the `cl_program` OpenCL program objects from the **.aocx** file.
3. Include the `clBuildProgram` function in your host application to create the program executable for the specified device.
   Below is an example host code on using `clCreateProgramWithBinary` to program an FPGA device:

```
size_t lengths[1];
unsigned char* binaries[1] ={NULL};
cl_int status[1];
cl_int error;
cl_program program;
const char options[] = "";

FILE *fp = fopen("program.aocx","rb");
fseek(fp,0,SEEK_END);
lengths[0] = ftell(fp);
binaries[0] = (unsigned char*)malloc(sizeof(unsigned char)*lengths[0]);
rewind(fp);
fread(binaries[0],lengths[0],1,fp);
fclose(fp);
```

```
program = clCreateProgramWithBinary(context,
                                    1,
                                    device_list,
                                    lengths,
                                    (const unsigned char **)binaries,
                                    status,
                                    &error);
clBuildProgram(program,1,device_list,options,NULL,NULL);
```

If the `clBuildProgram` function executes successfully, it returns `CL_SUCCESS`.

4. Create kernel objects from the program executable using the `clCreateKernelsInProgram` or `clCreateKernel` function.
5. Include the kernel execution function to instruct the host runtime to execute the scheduled kernel(s) on the FPGA.

    - To enqueue a command to execute an NDRange kernel, use `clEnqueueNDRangeKernel`.
    - To enqueue a single work-item kernel, use `clEnqueueTask`.

    You can load multiple FPGA programs into memory, which the host then uses to reprogram the FPGA as required.

For more information on these OpenCL host runtime application programming interface (API) calls, refer to the *OpenCL Specification version 1.0*.

Related Information

OpenCL Specification version 1.0

Programming Multiple FPGA Devices

If you install multiple FPGA devices in your system, you can direct the host runtime to program a specific FPGA device by modifying your host code.

Important: You may only program multiple FPGA devices from the *same* Custom Platform because the *AOCL_BOARD_PACKAGE_ROOT* environment variable points to the location of a single Custom Platform.

You can present up to 16 FPGA devices to your system in the following manner:

- Multiple FPGA accelerator boards, each consisting of a single FPGA.
- Multiple FPGAs on a single accelerator board that connects to the host system via a PCI Express (PCIe) switch.
- Combinations of the above.

The host runtime can load kernels onto each and every one of the FPGA devices. The FPGA devices can then operate in a parallel fashion.

1. **Probing the OpenCL FPGA Devices** on page 1-71
   The host must identify the number of OpenCL FPGA devices installed into the system.
2. **Querying Device Information** on page 1-71
   You can direct the host to query information on your OpenCL FPGA devices.
3. **Loading Kernels for Multiple FPGA Devices** on page 1-72
   If your system contains multiple FPGA devices, you can create specific `cl_program` objects for each FPGA and load them into the OpenCL runtime.

Probing the OpenCL FPGA Devices

The host must identify the number of OpenCL FPGA devices installed into the system.

1. To query a list of FPGA devices installed in your machine, invoke the `aocl diagnose` command.
2. To direct the host to identify the number of OpenCL FPGA devices, add the following lines of code to your host application:

```
//Get the platform
ciErrNum = oclGetPlatformID(&cpPlatform);

//Get the devices
ciErrNum = clGetDeviceIDs(cpPlatform,
                          CL_DEVICE_TYPE_ALL,
                          0,
                          NULL,
                          &ciDeviceCount);
cdDevices = (cl_device_id * )malloc(ciDeviceCount * sizeof(cl_device_id));
ciErrNum = clGetDeviceIDs(cpPlatform,
                          CL_DEVICE_TYPE_ALL,
                          ciDeviceCount,
                          cdDevices,
                          NULL);
```

For example, on a system with two OpenCL FPGA devices, `ciDeviceCount` has a value of 2, and `cdDevices` contains a list of two device IDs (`cl_device_id`).

Related Information

Querying the Device Name of Your FPGA Board (diagnose) on page 1-11

Querying Device Information

You can direct the host to query information on your OpenCL FPGA devices.

- To direct the host to output a list of OpenCL FPGA devices installed into your system, add the following lines of code to your host application:

```
char buf[1024];
for (unsigned i = 0; i < ciDeviceCount; i++);
{
    clGetDeviceInfo(cdDevices[i], CL_DEVICE_NAME, 1023, buf, 0);
    printf("Device %d: '%s'\n", i, buf);
}
```

When you query the device information, the host will list your FPGA devices in the following manner:

`Device <N>: <board_name>: <name_of_FPGA_board>`

Where:

<*N*> is the device number.

<*board_name*> is the board designation you use to target your FPGA device when you invoke the `aoc` command.

<*name_of_FPGA_board*> is the advertised name of the FPGA board.

For example, if you have two identical FPGA boards on your system, the host generates an output that resembles the following:

```
Device 0: board_1: Stratix V FPGA Board
Device 1: board_1: Stratix V FPGA Board
```

Note: The `clGetDeviceInfo` function returns the board type (for example, `board_1`) that the Altera Offline Compiler (AOC) lists on-screen when you invoke the `aoc --list-boards` command. If your accelerator board contains more than one FPGA, each device is treated as a "board" and is given a unique name.

Related Information

Listing the Available FPGA Boards in Your Custom Platform (--list-boards) on page 1-9

Loading Kernels for Multiple FPGA Devices

If your system contains multiple FPGA devices, you can create specific `cl_program` objects for each FPGA and load them into the OpenCL runtime.

The following host code demonstrates the usage of the `clCreateProgramWithBinary` and `createMulti-DeviceProgram` functions to program multiple FPGA devices:

```
cl_program createMultiDeviceProgram(cl_context context,
                                    const cl_device_id *device_list,
                                    cl_uint num_devices,
                                    const char *aocx_name);

// Utility function for loading file into Binary String
//
unsigned char* load_file(const char* filename, size_t *size_ret)
{
   FILE *fp = fopen(aocx_name,"rb");
   fseek(fp,0,SEEK_END);
   size_t len = ftell(fp);
   char *result = (unsigned char*)malloc(sizeof(unsigned char)*len);
   rewind(fp);
   fread(result,len,1,fp);
   fclose(fp);
   *size_ret = len;
   return result;
}

//Create a Program that is compiled for the devices in the "device_list"
//
cl_program createMultiDeviceProgram(cl_context context,
                                    const cl_device_id *device_list,
                                    cl_uint num_devices,
                                    const char *aocx_name)
{
    printf("creating multi device program %s for %d devices\n",
           aocx_name, num_devices);
    const unsigned char **binaries =
       (const unsigned char**)malloc(num_devices*sizeof(unsigned char*));
    size_t *lengths=(size_t*)malloc(num_devices*sizeof(size_t));
    cl_int err;

    for(cl_uint i=0; i<num_devices; i++)
    {
       binaries[i] = load_file(aocx_name,&lengths[i]);
       if (!binaries[i])
       {
          printf("couldn't load %s\n", aocx_name);
          exit(-1);
       }
    }

    cl_program p = clCreateProgramWithBinary(context,
                                             num_devices,
                                             device_list,
                                             lengths,
                                                         binaries,
                                                         NULL,
                                                         &err);
    free(lengths);
    free(binaries);

    if (err != CL_SUCCESS)
    {
       printf("Program Create Error\n");
    }
    return p;
}


// main program

main ()
{
   // Normal OpenCL setup
}
program = createMultiDeviceProgram(context,
                                   device_list,
                                   num_devices,
                                   "program.aocx");
clBuildProgram(program,num_devices,device_list,options,NULL,NULL);
```

Linking Your Host Application to the Khronos ICD Loader Library

The Altera SDK for OpenCL (AOCL) supports the OpenCL Installable Client Driver (ICD) extension from the Khronos Group. The OpenCL ICD extension allows you to have multiple OpenCL implementations on your system. With the OpenCL ICD Loader Library, you may choose from a list of installed platforms and execute OpenCL application programming interface (API) calls that are specific to your OpenCL implementation of choice.

In addition to the AOCL host runtime libraries, Altera supplies a version of the ICD Loader Library that supports the OpenCL Specification version 1.0. To use an ICD library from another vendor, consult the vendor's documentation on how to link to their ICD library.

Linking to the ICD Loader Library on Windows on page 1-73
To link your Windows OpenCL host application to the Installable Client Driver (ICD) Loader Library, modify the **Makefile** and set up the Altera Client Driver (ACD).

Linking to the ICD Loader Library on Linux on page 1-74
To link your Linux OpenCL host application to the Installable Client Driver (ICD) Loader Library, modify the **Makefile**. For Cyclone V SoC boards, you also have to create an **Altera.icd** file.

Linking to the ICD Loader Library on Windows

To link your Windows OpenCL host application to the Installable Client Driver (ICD) Loader Library, modify the **Makefile** and set up the Altera Client Driver (ACD).

Attention: For Windows systems, you must use the ICD in conjunction with the ACD. If the custom platform from your board vendor does not currently support ACD, you can set it up manually.

1. Prior to linking your host application to any Altera SDK for OpenCL (AOCL) host runtime libraries, link it to the OpenCL library by modifying the **Makefile**.
   A modified **Makefile** might include the following lines:

```
AOCL_COMPILE_CONFIG=$(shell aocl compile-config)
AOCL_LDFLAGS=$(shell aocl ldflags)
```

```
AOCL_LDLIBS=$(shell aocl ldlibs)

host_prog.exe : host_prog.obj
    link -nologo /OUT:host_prog.exe host_prog.obj $(AOCL_ LDFLAGS) OpenCL.lib $
(AOCL_LDLIBS)

host_prog.obj : host_prog.cpp
    cl /MD /Fohost_prog.obj -c host_prog.cpp $(AOCL_COMPILE_CONFIG)
```

2. If you need to manually set up ACD support for your Custom Platform, perform the following tasks:
   a. Consult with your board vendor to identify the libraries that the ACD requires. Alternatively, you may invoke the `aocl ldlibs` command and identify the libraries that your OpenCL application requires.
   b. Specify the libraries in the registry key **HKEY_LOCAL_MACHINE\SOFTWARE\Altera\OpenCL\Boards**. Enter one value for each library. Each value must include the path to the library as the string value, and a **DWORD** setting of 0.

      Attention: If your board vendor provides multiple libraries, you might need to load them in a particular order. Consult with your board vendor to determine the correct order to load the libraries. List the libraries in the registry in their loading order.

To enumerate board vendor-specific ICDs, the ICD Loader scans the values in the **HKEY_LOCAL_MACHINE\SOFTWARE\Altera\OpenCL\Boards** registry key. For each value in the key that has a DWORD value of 0, the ACD Loader opens the corresponding dynamic link library (DLL) specified in the key.

Consider the following registry key value:

[HKEY_LOCAL_MACHINE\SOFTWARE\Altera\OpenCL\Boards] "c:\\board_vendor a\my_board_mmd.dll"=dword:00000000

The ICD Loader scans this value, and then the ACD Loader opens the library **my_board_mmd.dll** from the **board_vendor a** folder.

Attention: If your host application fails to run while it is linking to the ICD, ensure that the **HKEY_LOCAL_MACHINE\SOFTWARE\Khronos\OpenCL\Vendors** registry key contains the following value:

[HKEY_LOCAL_MACHINE\SOFTWARE\Khronos\OpenCL\Vendors] "alteracl_icd.dll"=dword:00000000

Linking to the ICD Loader Library on Linux

To link your Linux OpenCL host application to the Installable Client Driver (ICD) Loader Library, modify the **Makefile**. For Cyclone V SoC boards, you also have to create an **Altera.icd** file.

1. Prior to linking your host application to any AOCL host runtime libraries, link it to the OpenCL library by modifying the **Makefile**.
   A modified **Makefile** might include the following lines:

```
AOCL_LDFLAGS=$(shell aocl ldflags)
AOCL_LDLIBS=$(shell aocl ldlibs)
```

```
host_prog : host_prog.o
g++ -o host_prog host_prog.o $(AOCL_LDFLAGS) -lOpenCL $(AOCL_LDLIBS)
```

2. For Cyclone V SoC boards, when you build the SD flash card image for your Custom Platform, create an **Altera.icd** file containing the text `libalteracl.so`. Store the **Altera.icd** file in the **/etc/OpenCL/vendors** directory of your Custom Platform.

   Refer to *Building an SD Flash Card Image* section of the *Altera Cyclone V SoC Development Kit Reference Platform Porting Guide* for more information.

Attention: If your host application fails to run while linking to the ICD, ensure that the file **/etc/OpenCL/vendors/Altera.icd** matches the file found in the directory that *ALTERAOCLSDKROOT* specifies. The environment variable *ALTERAOCLSDKROOT* points to the location of the AOCL installation. If the files do not match, or if it is missing from **/etc/OpenCL/vendors**, copy the **Altera.icd** file from *ALTERAOCLSDKROOT* to **/etc/OpenCL/vendors**.

Related Information

Building an SD Flash Card Image

Compiling Your OpenCL Kernel

The Altera SDK for OpenCL (AOCL) offers a list of compiler options that allows you to customize the kernel compilation process. For example, you can direct the Altera Offline Compiler (AOC) to target a specific FPGA board, generate reports, or implement optimization techniques.

Before you compile an OpenCL kernel, ensure that the environment variable *AOCL_BOARD_PACKAGE_ROOT* points to the location of the appropriate Custom Platform.

Attention: If you use the Altera Stratix V Network Reference Platform (s5_net), you must acquire and install the PLDA quick user datagram protocol (QuickUDP) intellectual property (IP) core license. Refer to the PLDA website for more information. If you use a Custom Platform that includes the QuickUDP IP core, refer to your board vendor's documentation for more information on the acquisition and installation of the QuickUDP IP license.

Caution: Improper installation of the QuickUDP IP license causes kernel compilation to fail with the following error message:

```
Error (292014): Can't find valid feature line for core PLDA
QUICKTCP (73E1_AE12) in current license.
```

Note that the error has no actual dependency on the TCP Hardware Stack QuickTCP IP from PLDA.

Compiling a Kernel for a Big-Endian System (--big-endian)

To direct the Altera Offline Compiler (AOC) to compile your OpenCL kernel and generate a hardware configuration file for use in a big-endian system (for example, the IBM POWER system), include the `--big-endian` option in the `aoc` command.

If you create an OpenCL kernel program that targets a big-endian architecture, you have to specify big-endian ordering for the host and global memories. If not, the AOC automatically defaults to little-endian ordering.

- At a command prompt, invoke the `aoc <your_kernel_filename>.cl --big-endian` command.

Compiling Your Kernel to Create Hardware Configuration File

You can compile an OpenCL kernel and create the hardware configuration file (that is, the Altera Offline Compiler Executable file (**.aocx**)) in a single step.

Altera recommends that you use this one-step compilation strategy under the following circumstances:

- After you optimize your kernel via the Altera SDK for OpenCL (AOCL) design flow, and you are now ready to create the **.aocx** file for deployment onto the FPGA.
- You have a simple kernel that does not require any optimization.

- To compile the kernel and generate the **.aocx** file in one step, invoke the `aoc <your_kernel_filename>.cl` command.

Compiling Your Kernel without Building Hardware (-c)

To direct the Altera Offline Compiler (AOC) to compile your OpenCL kernel and generate a Quartus II hardware design project without creating a hardware configuration file, include the `-c` option in your `aoc` command.

- At a command prompt, invoke the `aoc -c <your_kernel_filename>.cl` command.
  When you invoke the `aoc` command with the `-c` flag, the AOC compiles the kernel and creates the following files and directories:
  - The Altera Offline Compiler Object file (**.aoco**). The AOC creates the **.aoco** file in a matter of seconds to minutes.
  - A ***<your_kernel_filename>*** folder or subdirectory. It contains intermediate files that the Altera SDK for OpenCL (AOCL) uses to build the hardware configuration file necessary for FPGA programming.

Specifying the Location of Header Files (-I <directory>)

To add a directory to the list of directories that the Altera Offline Compiler (AOC) searches for header files during kernel compilation, include the `-I <directory>` option in your `aoc` command.

If the header files are in the same directory as your kernel, you do not need to include the `-I <directory>` option in your `aoc` command. The AOC automatically searches the current folder or directory for header files.

- At a command prompt, invoke the `aoc -I <directory> <your_kernel_filename>.cl` command.

  Caution: For Windows systems, ensure that your include path does not contain any trailing slashes. The AOC considers a trailing forward slash (/) or backward slash (\) as illegal.

  The AOC generates an error message if you invoke the `aoc` command in the following manner:

  ```
  aoc -I <drive>\<folder>\ ... \<subfolder>\
  <your_kernel_filename>.cl
  ```

  or

  ```
  aoc -I <drive>/<folder>/ ... /<subfolder>/
  <your_kernel_filename>.cl
  ```

  The correct way to specify the include path is as follows:

  ```
  aoc -I <drive>\<folder>\ ... \<subfolder>
  <your_kernel_filename>.cl
  ```

  or

  ```
  aoc -I <drive>/<folder>/ ... /<subfolder>
  <your_kernel_filename>.cl
  ```

Specifying the Name of an AOC Output File (-o <filename>)

To specify the name of an Altera Offline Compiler Object file (**.aoco**) or an Altera Offline Compiler Executable file (**.aocx**), include the `-o <filename>` option in your `aoc` command.

- If you implement the multistep compilation flow, specify the names of the output files in the following manner:
  1. To specify the name of the **.aoco** file that the Altera Offline Compiler (AOC) creates during an intermediate compilation step, invoke the `aoc -c -o <your_object_filename>.aoco <your kernel_filename>.cl` command.
  2. To specify the name of the **.aocx** file that the AOC creates during the final compilation step, invoke the `aoc -o <your_executable_filename>.aocx <your_object_filename>.aoco` command.
- If you implement the one-step compilation flow, specify the name of the **.aocx** file by invoking the `aoc -o <your_executable_filename>.aocx <your_kernel_filename>.cl` command.

Compiling a Kernel for a Specific FPGA Board (--board <board_name>)

To compile your OpenCL kernel for a specific FPGA board, include the `--board <board_name>` option in the `aoc` command.

Before you begin

To compile a kernel for a specific board in your Custom Platform, you must first set the environment variable *AOCL_BOARD_PACKAGE_ROOT* to point to the location of your Custom Platform.

Attention: If you want to program multiple FPGA devices, you may select board types that are available in the same Custom Platform because *AOCL_BOARD_PACKAGE_ROOT* only points to the location of one Custom Platform.

When you compile your kernel by including the `--board <board_name>` option in the `aoc` command, the Altera Offline Compiler (AOC) defines the preprocessor macro `AOCL_BOARD_<board_name>` to be 1, which allows you to compile device-optimized code in your kernel.

1. To obtain the names of the available FPGA boards in your Custom Platform, invoke the `aoc --list-boards` command.
   For example, the AOC generates the following output:

   ```
   Board List:
   FPGA_board_1
   ```

   where `FPGA_board_1` is the *<board_name>*.
2. To compile your OpenCL kernel for FPGA_board_1, invoke the `aoc --board FPGA_board_1 <your_kernel_filename>.cl` command.
   The AOC defines the preprocessor macro `AOCL_BOARD_FPGA_board_1` to be 1 and compiles kernel code that targets FPGA_board_1.

Tip: To readily identify compiled kernel files that target a specific FPGA board, Altera recommends that you rename the kernel binaries by including the `-o` option in the aoc command.

To target your kernel to FPGA_board_1 in the one-step compilation flow, invoke the following command:

```
aoc --board FPGA_board_1 <your_kernel_filename>.cl -o
<your_executable_filename>_FPGA_board_1.aocx
```

To target your kernel to FPGA_board_1 in the multistep compilation flow, perform the following tasks:

1. Invoke the following command to generate the Altera Offline Compiler Object File (**.aoco**):

   ```
   aoc -c --board FPGA_board_1 <your_kernel_filename>.cl
   -o <my_object_filename>_FPGA_board_1.aoco
   ```
2. Invoke the following command to generate the Altera Offline Compiler Executable file (**.aocx**):

   ```
   aoc --board FPGA_board_1
   <your_object_filename>_FPGA_board_1.aoco -o
   <your_executable_filename>_FPGA_board_1.aocx
   ```

If you have an accelerator board consisting of two FPGAs, each FPGA device has an equivalent "board" name (for example, board_fpga_1 and board_fpga_2). To target a **kernel_1.cl** to board_fpga_1 and a **kernel_2.cl** to board_fpga_2, invoke the following commands:

```
aoc --board board_fpga1 kernel_1.cl
```

```
aoc --board board_fpga2 kernel_2.cl
```

Related Information

Specifying the Name of an AOC Output File (-o <filename>) on page 1-77

Resolving Hardware Generation Fitting Errors during Kernel Compilation (--high-effort)

Sometimes, OpenCL kernel compilation fails during the hardware generation stage because the design fails to meet fitting constraints. In this case, recompile the kernel using the `--high-effort` option of the `aoc` command.

When kernel compilation fails because of a fitting constraint problem, the Altera Offline Compiler (AOC) displays the following error message:

```
Error: Kernel fit error, recommend using --high-effort.
Error: Cannot fit kernel(s) on device
```

- To overcome this problem, recompile your kernel by invoking the following command:
  `aoc --high-effort <your_kernel_filename>.cl`

  After you invoke the command, the AOC displays the following message:

  ```
  High-effort hardware generation selected, compile time may increase signifi-
  cantly.
  ```

The AOC will make three attempts to recompile your kernel and generate hardware. Modify your kernel if compilation still fails after the `--high-effort` attempt.

Defining Preprocessor Macros to Specify Kernel Parameters (-D <macro_name>)

The Altera Offline Compiler (AOC) supports preprocessor macros that allow you to pass macro definitions and compile code on a conditional basis.

- To pass a preprocessor macro definition to the AOC, invoke the `aoc -D <macro_name> <kernel_filename>.cl` command.
- To override the existing value of a defined preprocessor macro, invoke the `aoc -D <marco_name>=<value> <kernel_filename>.cl` command.
  Consider the following code snippet for the kernel `sum`:

```
#ifndef UNROLL_FACTOR
  #define UNROLL_FACTOR 1
#endif

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

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

  To override the `UNROLL_FACTOR` of 1 and set it to 4, invoke the `aoc -D UNROLL_FACTOR=4 sum.cl` command. Invoking this command is equivalent to replacing the line `#define UNROLL_FACTOR 1` with `#define UNROLL_FACTOR 4` in the `sum` kernel source code.
- To use preprocessor macros to control how the AOC optimizes your kernel without modifying your kernel source code, invoke the `aoc -o <hardware_filename>.aocx -D <macro_name>=<value> <kernel_filename>.cl`
  Where:

  `-o` is the AOC option you use to specify the name of the Altera Offline Compiler Executable file (**.aocx**) that the AOC generates.

  <hardware_filename> is the name of the **.aocx** file that the AOC generates using the preprocessor macro value you specify.

  Tip: To preserve the results from both compilations on your file system, compile your kernels as separate binaries by using the `-o` flag of the `aoc` command.

  For example, if you want to compile the same kernel multiple times with required work-group sizes of 64 and 128, you can define a `WORK_GROUP_SIZE` preprocessor macro for the kernel attribute `reqd_work_group_size`, as shown below:

```
__attribute__((reqd_work_group_size(WORK_GROUP_SIZE,1,1)))
__kernel void myKernel(...)
for (size_t i = 0; i < 1024; i++)
{
      // statements
}
```

  Compile the kernel multiple times by typing the following commands:

```
aoc -o myKernel_64.aocx -D WORK_GROUP_SIZE=64 myKernel.cl

aoc -o myKernel_128.aocx -D WORK_GROUP_SIZE=128 myKernel.cl
```

Generating Compilation Progress Report (-v)

To direct the Altera Offline Compiler (AOC) to report on the progress of a compilation, include the `-v` option in your `aoc` command.

- To direct the AOC to report on the progress of a full compilation, invoke the `aoc -v <`*`your_kernel_filename`*`>.cl` command.
  The AOC generates a compilation progress report similar to the following example:

```
aoc: Environment checks are completed successfully.
You are now compiling the full flow!!
aoc: Selected target board s5_net
aoc: Running OpenCL parser....
aoc: OpenCL parser completed successfully.
aoc: Compiling....
aoc: Linking with IP library ...
aoc: First stage compilation completed successfully.
aoc: Setting up project for CvP revision flow....
aoc: Hardware generation completed successfully.
```

- To direct the AOC to report on the progress of an intermediate compilation step that does not build hardware, invoke the `aoc -c -v <`*`your_kernel_filename`*`>.cl` command.
  The AOC generates a compilation progress report similar to the following example:

```
aoc: Environment checks are completed successfully.
aoc: Selected target board s5_net
aoc: Running OpenCL parser....
aoc: OpenCL parser completed successfully.
aoc: Compiling....
aoc: Linking with IP library ...
aoc: First stage compilation completed successfully.
aoc: To compile this project, run "aoc <your_kernel_filename>.aoco"
```

- To direct the AOC to report on the progress of a compilation for emulation, invoke the `aoc -march=emulator -v <`*`your_kernel_filename`*`>.cl` command.
  The AOC generates a compilation progress report similar to the following example:

```
aoc: Environment checks are completed successfully.
You are now compiling the full flow!!
aoc: Selected target board s5_net
aoc: Running OpenCL parser....ex
aoc: OpenCL parser completed successfully.
aoc: Compiling for Emulation ....
aoc: Emulator Compilation completed successfully.
Emulator flow is successful.
```

Related Information

- **Compiling Your Kernel without Building Hardware (-c)** on page 1-76
- **Emulating and Debugging Your OpenCL Kernel** on page 1-84

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

By default, the Altera Offline Compiler (AOC) estimates hardware resource usage during compilation . The AOC factors in the usage of external interfaces such as PCI Express (PCIe), memory controller, and direct memory access (DMA) engine in its calculations. During kernel compilation, the AOC generates an estimated resource usage summary in the ***<your_kernel_filename>*.log** file within the ***<your_kernel_filename>*** directory. To review the estimated resource usage summary on-screen, include the `--report` option in the `aoc` command.

You can review the estimated resource usage summary without performing a full compilation. To review the summary on-screen prior to generating the hardware configuration file, include the `-c` option in your aoc command.

- At a command prompt, invoke the `aoc -c <your_kernel_filename>.cl --report` command.
  The AOC generates an output similar to the following example:

```
aoc: Selected target board s5_ref

+--------------------------------------------------------------------+
; Estimated Resource Usage Summary                                   ;
+----------------------------------------+---------------------------+
; Resource                               + Usage                     ;
+----------------------------------------+---------------------------+
; Logic utilization                      ;   13%                     ;
; Dedicated logic registers              ;    5%                     ;
; Memory blocks                          ;   12%                     ;
; DSP blocks                             ;    0%                     ;
+----------------------------------------+---------------------------;
```

Related Information

Compiling Your Kernel without Building Hardware (-c) on page 1-76

Suppressing AOC Warning Messages (-W)

To suppress all warning messages, include the `-W` option in your `aoc` command.

- At a command prompt, invoke the `aoc -W <your_kernel_filename>.cl` command.

Converting AOC Warning Messages into Error Messages (-Werror)

To convert all warning messages into error messages, include the `-Werror` option in your `aoc` command.

- At a command prompt, invoke the `aoc -Werror <your_kernel_filename>.cl` command.

Adding Source References to Optimization Reports (-g)

Include the `-g` option in your `aoc` command to add source references to compilation reports.

When you compile a single work-item kernel, the Altera Offline Compiler (AOC) automatically generates an optimization report in the **<*your_kernel_filename*>.log** file in the **<*your_kernel_filename*>** subfolder or subdirectory. Adding source information such as line numbers and variable names in the optimization report allows you to pinpoint the locations of loop-carried dependencies in your kernel source code.

- To add source information in the optimization report, invoke the `aoc -g <your_kernel_filename>.cl` command.

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

The Altera Offline Compiler (AOC) cannot burst-interleave global memory across different memory types. You can disable burst-interleaving for all global memory banks of the same type and manage them manually by including the `--no-interleaving <global_memory_type>` option in your `aoc` command. Manual partitioning of memory buffers overrides the default burst-interleaved configuration of global memory.

Caution: The `--no-interleaving` option requires a global memory type parameter. If you do not specify a memory type, the AOC issues an error message.

- To direct the AOC to disable burst-interleaving for the default global memory, invoke the `aoc <your_kernel_filename>.cl --no-interleaving default` command.

  Your accelerator board might include multiple global memory types. To identify the default global memory type, refer to board vendor's documentation for your Custom Platform.
- For a heterogeneous memory system, to direct the AOC to disable burst-interleaving of a specific global memory type, perform the following tasks:
  1. Consult the **board_spec.xml** file of your Custom Platform for the names of the available global memory types (for example, DDR and quad data rate (QDR)).
  2. To disable burst-interleaving for one of the memory types (for example, DDR), invoke the `aoc <your_kernel_filename>.cl --no-interleaving DDR` command.
     The AOC enables manual partitioning for the DDR memory bank, and configures the other memory bank in a burst-interleaved fashion.
  3. To disable burst-interleaving for more than one type of global memory buffers, include a `--no-interleaving <global_memory_type>` option for each global memory type.
     For example, to disable burst-interleaving for both DDR and QDR, invoke the `aoc <your_kernel_filename>.cl --no-interleaving DDR --no-interleaving QDR` command.

Caution: Do not pass a buffer as kernel arguments that associate it with multiple memory technologies.

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

Include the `--const-cache-bytes <N>` flag in your `aoc` command to direct the Altera Offline Compiler (AOC) to configure the constant memory cache size (rounded up to the closest power of 2).

The default constant cache size is 16 kilobytes (kB).

- To configure the constant memory cache size, invoke the `aoc --const-cache-bytes <N> <your_kernel_filename>.cl` command, where *<N>* is the cache size in bytes.
  For example, to configure a 32 kB cache during compilation of the OpenCL kernel **myKernel.cl**, invoke the `aoc --const-cache-bytes 32768 myKernel.cl` command.

  Note: This argument has no effect if none of the kernels uses the `__constant` address space.

Relaxing the Order of Floating-Point Operations (--fp-relaxed)

Include the `--fp-relaxed` option in your `aoc` command to direct the Altera Offline Compiler (AOC) to relax the order of arithmetic floating-point operations using a balanced tree hardware implementation.

Implementing a balanced tree structure leads to more efficient hardware at the expense of numerical variation in results.

Caution: To implement this optimization control, your program must be able to tolerate small variations in the floating-point results.

- To direct the AOC to execute a balanced tree hardware implementation, invoke the `aoc --fp-relaxed <your_kernel_filename>.cl` command.

Reducing Floating-Point Rounding Operations (--fpc)

Include the `--fpc` option in your `aoc` command to direct the Altera Offline Compiler (AOC) to remove intermediary floating-point rounding operations and conversions whenever possible, and to carry additional bits to maintain precision.

Implementing this optimization control also changes the rounding mode. It rounds towards zero only at the end of a chain of floating-point arithmetic operations (that is, multiplications, additions, and subtractions).

- To direct the AOC to reduce the number of rounding operations, invoke the `aoc --fpc <your_kernel_filename>.cl` command.

Emulating and Debugging Your OpenCL Kernel

Use the Altera SDK for OpenCL (AOCL) emulator to assess the functionality of your kernel.

The AOCL Emulator generates an Altera Offline Compiler Executable file (**.aocx**) that executes on x86-64 Windows or Linux host. This feature allows you to emulate the functionality of your kernel and iterate on your design without executing it on the actual FPGA each time. For Linux platform, you can also use the Emulator to perform functional debug.

Caution: Emulation does not support cross-compilation to ARM processor. To run emulation on a design that targets an SoC, emulate on a non-SoC board (for example, ***ALTERAOCLSDKROOT*/board/s5_ref**). When you are satisfied with the emulation results, you may target your design on an SoC board for subsequent optimization steps.

1. **Modifying Channels Kernel Code for Emulation** on page 1-84
   To emulate applications with a channel that reads or writes to an I/O channel, modify your kernel to add a read or write channel that replaces the I/O channel, and make the source code that uses it is conditional.
2. **Compiling a Kernel for Emulation (-march=emulator)** on page 1-86
   To compile an OpenCL kernel for emulation, include the `-march=emulator` option in your `aoc` command.
3. **Emulating Your OpenCL Kernel** on page 1-86
   To emulate your OpenCL kernel, run the emulation Altera Offline Executable file (**.aocx**) on the platform on which you build your kernel.
4. **Debugging Your OpenCL Kernel on Linux** on page 1-87
   For Linux systems, you can direct the Altera SDK for OpenCL (AOCL) Emulator to run your OpenCL kernel in the debugger and debug it functionally as part of the host application.
5. **Limitations of the AOCL Emulator** on page 1-88
   The Altera SDK for OpenCL (AOCL) Emulator feature has some limitations.

Modifying Channels Kernel Code for Emulation

The Emulator emulates kernel-to-kernel channels. It does not support the emulation of I/O channels that interface with input or output features of your FPGA board. To emulate applications with a channel that reads or writes to an I/O channel, modify your kernel to add a read or write channel that replaces the I/O channel, and make the source code that uses it is conditional.

Before you begin

The Altera SDK for OpenCL (AOCL) does not set the `EMULATOR` macro definition. You must set it manually either from the command line or in the source code.

Consider the following kernel example:

```
channel unlong4 inchannel __attribute__((io("eth0_in")));

__kernel void send (int size)
{
    for (unsigned i=0; i < size; i++)
    {
        ulong4 data = read_channel_altera(inchannel);
        //statements
    }
}
```

To enable the Emulator to emulate a kernel with a channel that interfaces with an I/O channel, perform the following tasks:

1. Modify the kernel code in one of the following manner:
    - Add a matching `write_channel_altera` call such as the one shown below.

      ```
      #ifdef EMULATOR

      __kernel void io_in (__global char * restrict arr, int size)
      {
          for (unsigned i=0; i<size; i++)
          {
              ulong4 data = arr[i]; //arr[i] being an alternate data source
              write_channel_altera(inchannel, data);
          }
      }
      #endif
      ```

    - Replace the I/O channel access with a memory access, as shown below:

      ```
      __kernel void send (int size)
      {
          for (unsigned i=0; i < size; i++)
          {
              #ifndef EMULATOR

                  ulong4 data = read_channel_altera(inchannel);

              #else
                  ulong4 data = arr[i]; //arr[i] being an alternate data source

              #endif
              //statements
          }
      }
      ```
2. Modify the host application to create and start this conditional kernel during emulation.

Related Information

Implementing I/O Channels Using the io Channels Attribute on page 1-26

Compiling a Kernel for Emulation (-march=emulator)

To compile an OpenCL kernel for emulation, include the `-march=emulator` option in your `aoc` command.

Before you begin

- Before you perform kernel emulation, ensure that you install a Custom Platform from your board vendor for your FPGA accelerator boards. Ensure that the environment variable *AOCL_BOARD_PACKAGE_ROOT* points to the location of the Custom Platform. Alternatively, if your kernel targets a board from an Altera SDK for OpenCL (AOCL) Reference Platform, set *AOCL_BOARD_PACKAGE_ROOT* to the path of the Reference Platform (for example, ***ALTERAOCLSDK-ROOT*/board/<*Reference_Platform_name*>**).
- To emulate your kernels on Windows systems, you need the Microsoft linker and additional compilation time libraries. Verify that the *PATH* environment variable setting includes all the paths described in *Setting the Environment Variables for Windows* in the *Altera SDK for OpenCL Getting Started Guide*.

  The *PATH* environment variable setting must include the path to the **LINK.EXE** file in Microsoft Visual Studio.
- Ensure that your *LIB* environment variable setting includes the path to the Microsoft compilation time libraries.

  The compilation time libraries are available with Microsoft Visual Studio.
- Verify that the LD_LIBRARY_PATH environment variable setting includes all the paths described in *Setting the Environment Variables for Linux* in the *Altera SDK for OpenCL Getting Started Guide*.
- To create kernel programs executable on x86-64 host systems, invoke the `aoc -march=emulator <your_kernel_filename>.cl` command.
- To compile a kernel for emulation that targets a specific board, invoke the `aoc -march=emulator --board <board_name> <your_kernel_filename>.cl` command.
- For Linux systems, to direct the Altera Offline Compiler (AOC) to enable symbolic debug support for the debugger, invoke the `aoc -march=emulator -g <your_kernel_filename>.cl` command.

  Enabling AOC debug support allows you to pinpoint the origins of functional errors in your kernel source code.

Related Information

- **Adding Source References to Optimization Reports (-g)** on page 1-82
- **Compiling a Kernel for a Specific FPGA Board (--board <board_name>)** on page 1-77
- **Setting the Environment Variables for Windows**
- **Setting the Environment Variables for Linux**

Emulating Your OpenCL Kernel

To emulate your OpenCL kernel, run the emulation Altera Offline Executable file (**.aocx**) on the platform on which you build your kernel.

To emulate your kernel, perform the following steps:

1. Run the utility command `aocl linkflags` to find out which libraries are necessary for building a host application. The software lists the libraries for both emulation and nonemulation compilation flows.
2. Build a host application and link it to the libraries from Step 1.
3. Ensure that the ***<your_kernel_filename>*.aocx** file is in a location where the host can find it, preferably the current working directory.
4. To run the host application for emulation, invoke the `env CL_CONTEXT_EMULATOR_DEVICE_ALTERA=<number_of_devices> <host_application_filename>` command.
   This command specifies the number of identical emulation devices that the Emulator needs to provide.
5. If you change your host or kernel program and you want to test it, only recompile the modified host or kernel program and then rerun emulation.

Each invocation of the emulated kernel creates a shared library copy called ***<process_ID>*-libkernel.so** in a default temporary directory, where *<process_ID>* is a unique numerical value assigned to each emulation run. You may override the default directory by setting the *TMP* or *TEMP* environment variable on Windows, or setting *TMPDIR* on Linux.

Related Information

Displaying Information on OpenCL Host Runtime and MMD Libraries (link-config or linkflags) on page 1-68

Debugging Your OpenCL Kernel on Linux

For Linux systems, you can direct the Altera SDK for OpenCL (AOCL) Emulator to run your OpenCL kernel in the debugger and debug it functionally as part of the host application. The debugging feature allows you to debug the host and the kernel seamlessly. You can step through your code, set breakpoints, and examine and set variables.

Prior to debugging your kernel, you must perform the following tasks:

1. During program execution, the debugger cannot step from the host code to the kernel code. You must set a breakpoint before the actual kernel invocation by adding these lines:
   a. `break <your_kernel>`

      This line sets a breakpoint before the kernel.
   b. `continue`

      If you have not begun debugging your host, then type `start` instead.
2. The kernel is loaded as a shared library immediately before the host loads the kernels. The debugger does not recognize the kernel names until the host actually loads the kernel functions. As a result, the debugger will generate the following warning for the breakpoint you set before the execution of the first kernel:

   ```
   Function "<your_kernel>" not defined.
   Make breakpoint pending on future shared library load? (y or [n])
   ```

   Answer `y`. After initial program execution, the debugger will recognize the function and variable names, and line number references for the duration of the session.

Caution: The Emulator uses the OpenCL runtime to report some error details. For emulation, the runtime uses a default print out callback when you initialize a context via the `clCreateContext` function.

Note: Kernel debugging is independent of host debugging. Debug your host code in existing tools such as Microsoft Visual Studio Debugger for Windows and GNU Project Debugger (GDB) for Linux.

To compile your OpenCL kernel for debugging, perform the following steps:

1. To generate an Altera Offline Compiler Executable file (**.aocx**) for debugging that targets a specific accelerator board, invoke the `aoc -march=emulator -g <`*`your_kernel_filename`*`>.cl --board <`*`board_name`*`>` command.

   Attention: Specify the name of your FPGA board when you run your host application. To verify the name of the target board for which you compile your kernel, invoke the `aoc -march=emulator -g -v <`*`your_kernel_filename`*`>.cl` command. The AOC will display the name of the target FPGA board.
2. Run the utility command `aocl linkflags` to find out the additional libraries necessary to build a host application that supports kernel debugging.
3. Build a host application and link it to the libraries from Step 2.
4. Ensure that the **<*your_kernel_filename*>.aocx** file is in a location where the host can find it, preferably the current working directory.
5. To run the application, invoke the command `env CL_CONTEXT_EMULATOR_DEVICE_ALTERA=<`*`number_of_devices`*`> gdb --args <`*`your_host_program_name`*`>`, where <*number_of_devices*> is the number of identical emulation devices that the Emulator needs to provide.
6. If you change your host or kernel program and you want to test it, only recompile the modified host or kernel program and then rerun the debugger.

Related Information

- **Adding Source References to Optimization Reports (-g)** on page 1-82
- **Compiling a Kernel for a Specific FPGA Board (--board <board_name>)** on page 1-77
- **Generating Compilation Progress Report (-v)** on page 1-81
- **Displaying Information on OpenCL Host Runtime and MMD Libraries (link-config or linkflags)** on page 1-68

Limitations of the AOCL Emulator

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

1. Execution model

   The Emulator supports the same compilation modes as the FPGA variant. As a result, you must call the `clCreateProgramBinary` function to create `cl_program` objects for emulation.

2. Concurrent execution

   Modeling of concurrent kernel executions has limitations. During execution, the Emulator does not actually run interacting work-items in parallel. Therefore, some concurrent execution behaviors, such as different kernels accessing global memory without a barrier for synchronization, might generate inconsistent emulation results between executions.

3. Kernel performance

   The Altera Offline Compiler Executable file (**.aocx**) that you generate for emulation does not include any optimizations. Therefore, it might execute at a significantly slower speed than what an optimized kernel might achieve. In addition, because the Emulator does not implement actual parallel execution, the execution time multiplies with the number of work-items that the kernel executes.

4. The Emulator executes the host runtime and the kernels in the same address space. Certain pointer or array usages in your host application might cause the kernel program to fail, and vice versa. Example usages include indexing external allocated memory and writing to random pointers. You may use memory leak detection tools such as Valgrind to analyze your program. However, the host might encounter a fatal error caused by out-of-bounds write operations in your kernel, and vice versa.

5. Emulation of channel behavior has limitations, especially for conditional channel operations where the kernel does not call the channel operation in every loop iteration. In these cases, the Emulator might execute channel operations in a different order than on the hardware.

Profiling Your OpenCL Kernel

The Altera SDK for OpenCL (AOCL) Profiler measures and reports performance data collected during OpenCL kernel execution on the FPGA. The AOCL Profiler relies on performance counters to gather kernel performance data. You can then review performance data in the profiler GUI.

1. **Instrumenting the Kernel Pipeline with Performance Counters (--profile)** on page 1-89
   To instrument the OpenCL kernel pipeline with performance counters, include the `--profile` option of the `aoc` command when you compile your kernel.
2. **Launching the AOCL Profiler GUI (report)** on page 1-90
   You can use the Altera SDK for OpenCL (AOCL) Profiler `report` utility command to launch the Profiler GUI.

Instrumenting the Kernel Pipeline with Performance Counters (--profile)

To instrument the OpenCL kernel pipeline with performance counters, include the `--profile` option of the `aoc` command when you compile your kernel.

Attention: Instrumenting the Verilog code with performance counters increases hardware resource utilization (that is, increases FPGA area usage) and typically decreases performance.

- To instrument the Verilog code in the **<*your_kernel_filename*>.aocx** Altera Offline Compiler Executable file with performance counters, invoke the `aoc --profile <your_kernel_filename>.cl` command.
- Run your host application from a local disk to execute the **<*your_kernel_filename*>.aocx** Altera Offline Compiler Executable file on your FPGA. During kernel execution, the performance counters throughout the kernel pipeline collects profile information. The host saves the information in a **profile.mon** monitor description file in your current working directory.

  Caution: Because of slow network disk accesses, running the host application from a networked directory might introduce delays between kernel executions. These delays might increase the overall execution time of the host application. In addition, they might introduce delays between kernel launches while the runtime stores profile output data to disk.

Launching the AOCL Profiler GUI (report)

You can use the Altera SDK for OpenCL (AOCL) Profiler `report` utility command to launch the Profiler GUI. The Profiler GUI allows you to view kernel performance data statistics that the AOCL Profiler collects during kernel execution.

The AOCL Profiler stores performance data in a **profile.mon** file in your current working directory.

1. To launch the Profiler GUI, invoke the `aocl report <your_kernel_filename>.aocx profile.mon` utility command.

Conclusion

You have now familiarized yourself with the Altera SDK for OpenCL design flow and the tools available to help you achieve your design goals. For more information on the support statuses of the OpenCL application programming interfaces (APIs) and programming language, refer to *Appendix A: Support Statuses of OpenCL Features*.

For in-depth information on optimizing your OpenCL kernel to maximize performance, refer to the *Altera SDK for OpenCL Best Practices Guide*.

Related Information

Altera SDK for OpenCL Best Practices Guide

Document Revision History

Table 1-3: Document Revision History of the Altera SDK for OpenCL Programming Guide

Date	Version	Changes
May 2015	15.0.0	• In *Guidelines for Naming the Kernel*, added entry that advised against naming an OpenCL kernel **kernel.cl**. • In *Instrumenting the Kernel Pipeline with Performance Counters (--profile)*, specified that you should run the host application from a local disk to avoid potential delays caused by slow network disk accesses. • In *Emulating and Debugging Your OpenCL Kernel*, modified Caution note to indicate that you must emulate a design targeting an SoC on a non-SoC board. • In *Emulating Your OpenCL Kernel*, updated command to run the host application and added instruction for overriding default temporary directory containing **<*process_ID*>-libkernel.so**. • Introduced the `--high-effort` `aoc` command flag in *Resolving Hardware Generation Fitting Errors during Kernel Compilation*. • In *Enabling Double Precision Floating-Point Operations*, introduced the `OPENCL EXTENSION` pragma for enabling double precision floating-point operations. • Introduced OpenCL pipes support. Refer to *Implementing OpenCL Pipes* (and subsequent subtopics) and *Creating a Pipe Object in Your Host Application* for more information. • In *AOCL Channels Extension: Restrictions*, added code examples to demonstrate how to statically index into arrays of channel IDs. • In *Multiple Host Threads*, added recommendation for synchronizing OpenCL host function calls in a multi-threaded host application. • Introduced ICD and ACD support. Refer to *Linking Your Host Application to the Khronos ICD Loader Library* for more information. • Introduced `clGetBoardExtenstionFunctionAddressAltera` for referencing user-accessible functions. Refer to *Accessing Custom Platform-Specific Functions* for more information.

Date	Version	Changes
December 2014	14.1.0	• Reorganized information flow. Information is now presented based on the tasks you might perform using the Altera SDK for OpenCL (AOCL) or the Altera RTE for OpenCL. • Removed information pertaining to the `--util <N>` and `-O3` Altera Offline Compiler (AOC) options. • Added the following information on PLDA QuickUDP IP core licensing in *Compiling Your OpenCL Kernel*: **1.** A PLDA QuickUDP IP core license is required for the Stratix V Network Reference Platform or a Custom Platform that uses the QuickUDP IP core. **2.** Improper installation of the QuickUDP IP core licence causes compilation to fail with an error message that refers to the QuickTCP IP core. • Added reminder that conditionally shifting a large shift register is not recommended. • Removed the *Emulating Systems with Multiple Devices* section. A new `env CL_CONTEXT_EMULATOR_DEVICE_ALTERA=<number_of_devices>` command is now available for emulating multiple devices. • Removed language support limitation from the *Limitations of the AOCL Emulator* section.

Date	Version	Changes
June 2014	14.0.0	• Removed the `--estimate-throughput` and `--sw-dimm-partition` AOC options • Added the `-march=emulator`, `-g`, `--big-endian`, and `--profile` AOC options • `--no-interleaving` needs <*global_memory_type*> argument • `-fp-relaxed=true` is now `--fp-relaxed` • `-fpc=true` is now `--fpc` • For non-SoC devices, `aocl diagnostic` is now `aocl diagnose` and `aocl diagnose <device_name>` • `program` and `flash` need <*device_name*> arguments • Added *Identifying the Device Name of Your FPGA Board* • Added *AOCL Profiler Utility* • Added *AOCL Channels Extension* and associated subsections • Added *Attributes for Channels* • Added *Match Data Layouts of Host and Kernel Structure Data Types* • Added *Register Inference* and *Shift Register Inference* • Added *Channels and Multiple Command Queues* • Added *Shared Memory Accesses for OpenCL Kernels Running on SoCs* • Added *Collecting Profile Data During Kernel Execution* • Added *Emulate and Debug Your OpenCL Kernel* and associated subsections • Updated *AOC Kernel Compilation Flows* • Updated *-v* • Updated *Host Binary Requirement* • Combined *Partitioning Global Memory Accesses* and *Partitioning Heterogeneous Global Memory Accesses* into the section *Partitioning Global Memory Accesses* • Updated *AOC Allocation Limits* in *Appendix A* • Removed `max_unroll_loops`, `max_share_resources`, `num_share_resources`, and `task` kernel attributes • Added `packed`, and `aligned(<N>)` kernel attributes

Date	Version	Changes
December 2013	13.1.1	• Removed the section -*W and* -*Werror*, and replaced it with two sections: -*W* and -*Werror*. • Updated the following contents to reflect multiple devices support: • The figure *The AOCL FPGA Programming Flow*. • *--list-boards* section. • *-board <board_name>* section. • *AOCL Utilities for Managing an FPGA Board* section. • Added the subsection *Programming Multiple FPGA Devices* under *FPGA Programming*. • The following contents were added to reflect heterogeneous global memory support: • *--no-interleaving* section. • `buffer_location` kernel attribute under *Kernel Pragmas and Attributes*. • *Partitioning Heterogeneous Global Memory Accesses* section. • Modified support status designations in *Appendix: Support Statuses of OpenCL Features*. • Removed information on OpenCL programming language restrictions from the section *OpenCL Programming Language Implementation*, and presented the information in a new section titled *OpenCL Programming Language Restrictions*.

Date	Version	Changes
Novembe r 2013	13.1.0	• Reorganized information flow. • Updated and renamed *Altera SDK for OpenCL Compilation Flow* to *AOCL FPGA Programming Flow*. • Added figures *One-Step AOC Compilation Flow* and *Two-Step AOC Compilation Flow*. • Updated the section *Contents of the AOCL Version 13.1*. • Removed the following sections: • *OpenCL Kernel Source File Compilation*. • *Using the Altera Offline Kernel Compiler*. • *Setting Up Your FPGA Board*. • *Targeting a Specific FPGA Board*. • *Running Your OpenCL Application*. • *Consolidating Your Kernel Source Files*. • *Aligned Memory Allocation*. • *Programming the FPGA Hardware*. • *Programming the Flash Memory of an FPGA*. • Updated and renamed *Compiling the OpenCL Kernel Source File* to *AOC Compilation Flows*. • Renamed *Passing File Scope Structures to OpenCL Kernels* to *Use Structure Arguments in OpenCL Kernels*. • Updated and renamed *Augmenting Your OpenCL Kernel by Specifying Kernel Attributes and Pragmas* to *Kernel Pragmas and Attributes*. • Renamed *Loading Kernels onto an FPGA* to *FPGA Programming*. • Consolidated *Compiling and Linking Your Host Program*, *Host Program Compilation Settings*, and *Library Paths and Links* into a single section. • Inserted the section *Preprocessor Macros*. • Renamed *Optimizing Global Memory Accesses* to *Partitioning Global Memory Accesses*.

Date	Version	Changes
June 2013	13.0 SP1.0	• Added the section *Setting Up Your FPGA Board*. • Removed the subsection *Specifying a Target FPGA Board* under *Kernel Programming Considerations*. • Inserted the subsections *Targeting a Specific FPGA Board* and *Generating Compilation Reports* under *Compiling the OpenCL Kernel Source File*. • Renamed *File Scope __constant Address Space Qualifier* to *__constant Address Space Qualifiers*, and inserted the following subsections: • *Function Scope __constant Variables*. • *File Scope __constant Variables*. • *Points to __constant Parameters from the Host*. • Inserted the subsection *Passing File Scope Structures to OpenCL Kernels* under *Kernel Programming Considerations*. • Renamed *Modifying Your OpenCL Kernel by Specifying Kernel Attributes and Pragmas* to *Augmenting Your OpenCL Kernel by Specifying Kernel Attributes and Pragmas*. • Updated content for the `unroll` pragma directive in the section *Augmenting Your OpenCL Kernel by Specifying Kernel Attributes and Pragmas*. • Inserted the subsections *Out-of-Order Command Queues* and *Modifying Host Program for Structure Parameter Conversion* under *Host Programming Considerations*. • Updated the sections *Loading Kernels onto an FPGA Using clClreateProgram-WithBinary* and *Aligned Memory Allocation*. • Updated flash programming instructions. • Renamed *Optional Extensions* in *Appendix B* to *Atomic Functions*, and updated its content. • Removed *Platform Layer and Runtime Implementation* from *Appendix B*.
May 2013	13.0.1	• Explicit memory fence functions are now supported; the entry is removed from the table OpenCL Programming Language Implementation. • Updated the section Programming the Flash Memory of an FPGA. • Added the section *Modifying Your OpenCL Kernel by Specifying Kernel Attributes and Pragmas* to introduce kernel attributes and pragmas that can be implemented to optimize kernel performance. • Added the section Optimizing Global Memory Accesses to discuss data partitioning. • Removed the section *Programming the FPGA with the aocl program Command* from Appendix A.

Date	Version	Changes
May 2013	13.0.0	• Updated compilation flow. • Updated kernel compiler commands. • Included Altera SDK for OpenCL Utility commands. • Added the section *OpenCL Programming Considerations*. • Updated flash programming procedure and moved it to *Appendix A*. • Included a new `clCreateProgramWithBinary` FPGA hardware programming flow. • Moved the hostless `clCreateProgramWithBinary` hardware programming flow to *Appendix A* under the title *Programming the FPGA with the aocl program Command*. • Moved updated information on allocation limits and OpenCL language support to *Appendix B*.
Novembe r 2012	12.1.0	Initial release.

Support Statuses of OpenCL Features A

2015.05.04

OCL002-15.0.0 Subscribe Send Feedback

The Altera SDK for OpenCL (AOCL) supports the OpenCL Specification version 1.0. The AOCL host runtime conforms with the OpenCL platform layer and application programming interface (API), with clarifications and exceptions.

The following sections outline the support statuses of the OpenCL features described in the *OpenCL Specification version 1.0*.

Related Information

OpenCL Specification version 1.0

OpenCL Programming Language Implementation

OpenCL is based on C99 with some limitations. Section 6 of the *OpenCL Specification version 1.0* describes the OpenCL C programming language. The Altera SDK for OpenCL (AOCL) conforms with the OpenCL C programming language with clarifications and exceptions. The table below summarizes the support statuses of the features in the OpenCL programming language implementation.

Attention: The support status "●" means that the feature is supported, and there might be a clarification for the supported feature in the Notes column. The support status "○" means that the feature is supported with exceptions identified in the Notes column. A feature that is not supported by the AOCL is identified with an "X". OpenCL programming language implementations that are supported with no additional clarifications are not shown.

© 2015 Altera Corporation. All rights reserved. ALTERA, ARRIA, CYCLONE, ENPIRION, MAX, MEGACORE, NIOS, QUARTUS and STRATIX words and logos are trademarks of Altera Corporation and registered in the U.S. Patent and Trademark Office and in other countries. All other words and logos identified as trademarks or service marks are the property of their respective holders as described at www.altera.com/common/legal.html. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

ISO 9001:2008 Registered

Section	Feature	Support Status	Notes
6.1.1	*Built-in Scalar Data Types*		
	double precision float	○	Preliminary support for all double precision float built-in scalar data type. This feature might not conform with the OpenCL Specification version 1.0. Currently, the following double precision floating-point functions conform with the OpenCL Specification version 1.0: add / subtract / multiply / divide / ceil / floor / rint / trunc / fabs / fmax / fmin / sqrt / rsqrt / exp / exp2 / exp10 / log / log2 / log10 / sin / cos / asin / acos
	half precision float	X	Support for scalar addition, subtraction and multiplication. Support for conversions to and from single-precision floating point. This feature might not conform with the OpenCL Specification version 1.0.
6.1.2	*Built-in Vector Data Types*	○	Preliminary support for vectors with three elements. Three-element vector support is a supplement to the OpenCL Specification version 1.0.
6.1.3	*Built-in Data Types*	X	
6.1.4	*Reserved Data Types*	X	
6.1.5	*Alignment of Types*	●	All scalar and vector types are aligned as required (vectors with three elements are aligned as if they had four elements).
6.2.1	*Implicit Conversions*	●	Refer to Section 6.2.6: *Usual Arithmetic Conversions* in the *OpenCL Specification version 1.2* for an important clarification of implicit conversions between scalar and vector types.
6.2.2	*Explicit Casts*	●	The AOCL allows scalar data casts to a vector with a different element type.
6.5	*Address Space Qualifiers*	○	Function scope `__constant` variables are not supported.
6.6	*Image Access Qualifiers*	X	
6.7	*Function Qualifiers*		
6.7.2	*Optional Attribute Qualifiers*	●	Refer to the *Altera SDK for OpenCL Best Practices Guide* for tips on using `reqd_work_group_size` to improve kernel performance. The AOCL parses but ignores the `vec_type_hint` and `work_group_size_hint` attribute qualifiers.

Section	Feature	Support Status	Notes
6.9	*Preprocessor Directives and Macros*		
	`#pragma` directive: `#pragma unroll`	●	The Altera Offline Compiler (AOC) supports only `#pragma unroll`. You may assign an integer argument to the unroll directive to control the extent of loop unrolling. For example, `#pragma unroll 4` unrolls four iterations of a loop. By default, an unroll directive with no unroll factor causes the AOC to attempt to unroll the loop fully. Refer to the *Altera SDK for OpenCL Best Practices Guide* for tips on using `#pragma unroll` to improve kernel performance.
	`__ENDIAN_LITTLE__` defined to be value `1`	●	The target FPGA is little-endian.
	`__IMAGE_SUPPORT__`	X	`__IMAGE_SUPPORT__` is undefined; the AOCL does not support images.
6.10	*Attribute Qualifiers*—The AOC parses attribute qualifiers as follows:		
6.10.2	*Specifying Attributes of Functions*—Structure-type kernel arguments	X	Convert structure arguments to a pointer to a structure in global memory.
6.10.3	*Specifying Attributes of Variables*—`endian`	X	
6.10.4	*Specifying Attributes of Blocks and Control-Flow-Statements*	X	
6.10.5	*Extending Attribute Qualifiers*	●	The AOC can parse attributes on various syntactic structures. It reserves some attribute names for its own internal use. Refer to the *Altera SDK for OpenCL Best Practices Guide* for tips on how to optimize kernel performance using these kernel attributes.
6.11.2	*Math Functions*		
	built-in math functions	○	Preliminary support for built-in math functions for double precision float. These functions might not conform with the OpenCL Specification version 1.0.
	built-in `half_` and `native_` math functions	○	Preliminary support for built-in `half_` and `native_` math functions for double precision float. These functions might not conform with the OpenCL Specification version 1.0.

Section	Feature	Support Status	Notes
6.11.5	*Geometric Functions*	○	Preliminary support for built-in geometric functions for double precision float. These functions might not conform with the OpenCL Specification version 1.0. Refer to *Argument Types for Built-in Geometric Functions* for a list of built-in geometric functions supported by the AOCL.
6.11.8	*Image Read and Write Functions*	X	
6.11.9	*Synchronization Functions*—the barrier synchronization function	○	Clarifications and exceptions: If a kernel specifies the `reqd_work_group_size` or `max_work_group_size` attribute, barrier supports the corresponding number of work-items. If neither attribute is specified, a barrier is instanti-ated with a default limit of 256 work-items. The work-item limit is the maximum supported work-group size for the kernel; this limit is enforced by the runtime.
6.11.11	*Async Copies from Global to Local Memory, Local to Global Memory, and Prefetch*	○	The implementation is naive: Work-item (0,0,0) performs the copy and the `wait_group_events` is implemented as a barrier. If a kernel specifies the `reqd_work_group_size` or `max_work_group_size` attribute, `wait_group_events` supports the corresponding number of work-items. If neither attribute is specified, `wait_group_events` is instantiated with a default limit of 256 work-items.
Additional built-in vector functions from the *OpenCL Specification version 1.2* Section 6.12.12: *Miscella-neous Vector Functions*:			
	`vec_step`	●	
	`shuffle`	●	
	`shuffle2`	●	
OpenCL Specification version 1.2 Section 6.12.13: *printf*		○	Preliminary support. This feature might not conform with the OpenCL Specification version 1.0. See below for details.

Section	Feature	Support Status	Notes
The `printf` function in OpenCL has syntax and features similar to the `printf` function in C99, with a few exceptions. For details, refer to the *OpenCL Specification version 1.2*. To use a `printf` function, there are no requirements for special compilation steps, buffers, or flags. You can compile kernels that include `printf` instructions with the usual `aoc` command. During kernel execution, `printf` data is stored in a global `printf` buffer that the AOC allocates automatically. The size of this buffer is 64 kB; the total size of data arguments to a `printf` call should not exceed this size. When kernel execution completes, the contents of the `printf` buffer are printed to standard output. Buffer overflows are handled seamlessly; `printf` instructions can be executed an unlimited number of times. However, if the `printf` buffer overflows, kernel pipeline execution stalls until the host reads the buffer and prints the buffer contents. Because `printf` functions store their data into a global memory buffer, the performance of your kernel will drop if it includes such functions. There are no usage limitations on `printf` functions. You can use `printf` instructions inside `if-then-else` statements, loops, etc. A kernel can contain multiple `printf` instructions executed by multiple work-items. Format string arguments and literal string arguments of `printf` calls are transferred to the host system from the FPGA using a special memory region. This memory region can overflow if the total size of the `printf` string arguments is large (3000 characters or less is usually safe in a typical OpenCL application). If there is an overflow, the error message `cannot parse auto-discovery string at byte offset 4096` is printed during host program execution. Output from `printf` is never intermixed, even though work-items may execute `printf` functions concurrently. However, the order of concurrent `printf` execution is not guaranteed. In other words, `printf` outputs might not appear in program order if the `printf` instructions are in concurrent datapaths.			

Related Information

- **Altera SDK for OpenCL Best Practices Guide**
- **OpenCL Specification version 1.2**
- **Argument Types for Built-in Geometric Functions** on page 2-8

OpenCL Programming Language Restrictions

The Altera SDK for OpenCL (AOCL) conforms with the OpenCL Specification restrictions on specific programming language features, as described in section 6.8 of the *OpenCL Specification version 1.0*.

Warning: The Altera Offline Compiler (AOC) does not enforce restrictions on certain disallowed programming language features. Ensure that your kernel code does not contain features that the OpenCL Specification version 1.0 does not support.

Feature	Support Status	Notes
pointer assignments between address spaces	●	Arguments to `__kernel` functions declared in a program that are pointers must be declared with the `__global`, `__constant`, or `__local` qualifier. The AOC enforces the OpenCL restriction against pointer assignments between address spaces.
pointers to functions	X	The AOC does not enforce this restriction.
structure-type kernel arguments	X	Convert structure arguments to a pointer to a structure in global memory.
images	X	The AOCL does not support images.
bit fields	X	The AOC does not enforce this restriction.
variable length arrays and structures	X	
variable macros and functions	X	
C99 headers	X	
`extern`, `static`, `auto`, and `register` storage-class specifiers	X	The AOC does not enforce this restriction.
predefined identifiers	●	Use the `-D` option of the `aoc` command to provide preprocessor symbol definitions in your kernel code.
recursion	X	The AOC does not enforce this restriction.
irreducible control flow	X	The AOC does not enforce this restriction.
writes to memory of built-in types less than 32 bits in size	○	Store operations less than 32 bits in size might result in lower memory performance.
declaration of arguments to `__kernel` functions of type `event_t`	X	The AOC does not enforce this restriction.
elements of a `struct` or a `union` belonging to different address spaces	X	The AOC does not enforce this restriction. **Warning:** Assigning elements of a `struct` or a `union` to different address spaces might cause a fatal error.

OpenCL C Programming Language Restrictions for Pipes

The Altera SDK for OpenCL (AOCL) offers preliminary support of OpenCL pipes.The following table lists the support statuses of pipe-specific OpenCL C programming language implementations, as described in the *OpenCL Specification version 2.0*

Attention: The support status "●" means that the feature is supported. There might be a clarification for the supported feature in the Notes column. A feature that is not supported by the AOCL is identified with an "X".

Table A-1: Support Statuses of Built-in Pipe Read and Write Functions

Details of the built-in pipe read and write functions are available in section 6.13.16.2 of the *OpenCL Specification version 2.0*.

Function	Support Status
`int read_pipe (pipe gentype p, gentype *ptr)`	●
`int write_pipe (pipe gentype p, const gentype *ptr)`	●
`int read_pipe (pipe gentype p, reserve_id_t reserve_id, uint index, gentype *ptr)`	X
`int write_pipe (pipe gentype p, reserve_id_t reserve_id, uint index, const gentype *ptr)`	X
`reserve_id_t reserve_read_pipe (pipe gentype p, uint num_packets)`	X
`reserve_id_t reserve_write_pipe (pipe gentype p, uint num_packets)`	
`void commit_read_pipe (pipe gentype p, reserve_id_t reserve_id)`	X
`void commit_write_pipe (pipe gentype p, reserve_id_t reserve_id)`	
`bool is_valid_reserve_id (reserve_id_t reserve_id)`	X

Table A-2: Support Statuses of Built-in Work-Group Pipe Read and Write Functions

Details of the built-in pipe read and write functions are available in section 6.13.16.3 of the *OpenCL Specification version 2.0*.

Function	Support Status
`reserve_id_t work_group_reserve_read_pipe (pipe gentype p, uint num_packets)`	X
`reserve_id_t work_group_reserve_write_pipe (pipe gentype p, uint num_packets)`	
`void work_group_commit_read_pipe (pipe gentype p, reserve_id_t reserve_id)`	X
`void work_group_commit_write_pipe (pipe gentype p, reserve_id_t reserve_id)`	

Table A-3: Support Statuses of Built-in Pipe Query Functions

Details of the built-in pipe query functions are available in section 6.13.16.4 of the *OpenCL Specification version 2.0*.

Function	Support Status
`uint get_pipe_num_packets (pipe gentype p)`	X
`uint get_pipe_max_packets (pipe gentype p)`	X

Related Information

OpenCL Specification version 2.0 (C Language)

Argument Types for Built-in Geometric Functions

The Altera SDK for OpenCL (AOCL) supports scalar and vector argument built-in geometric functions with certain limitations.

Function	Argument Type	
	float	double
`cross`	●	●
`dot`		●
`distance`		●
`length`		●
`normalize`		●
`fast_distance`		—
`fast_length`		—
`fast_normalize`		—

Numerical Compliance Implementation

Section 7 of the *OpenCL Specification version 1.0* describes features of the C99 and IEEE 754 standards that OpenCL-compliant devices must support. The Altera SDK for OpenCL (AOCL) operates on 32-bit and 64-bit floating-point values in IEEE Standard 754-2008 format, but not all floating-point operators have been implemented.

The table below summarizes the implementation statuses of the floating-point operators:

Section	Feature	Support Status	Notes
7.1	*Rounding Modes*	○	Conversion between integer and single and half precision floating-point types support all rounding modes. Conversions between integer and double precision floating-point types support all rounding modes on a preliminary basis. This feature might not conform with the OpenCL Specification version 1.0.

Section	Feature	Support Status	Notes
7.2	*INF, NaN and Denormalized Numbers*	○	Infinity (INF) and Not a Number (NaN) results for single precision operations are generated in a manner that conforms with the OpenCL Specification version 1.0. Most operations that handle denormalized numbers are flushed prior to and after a floating-point operation. Preliminary support for double precision floating-point operation. This feature might not conform with the OpenCL Specification version 1.0.
7.3	*Floating-Point Exceptions*	X	
7.4	*Relative Error as ULPs*	○	Single precision floating-point operations conform with the numerical accuracy requirements for an embedded profile of the OpenCL Specification version 1.0. Preliminary support for double precision floating-point operation. This feature might not conform with the OpenCL Specification version 1.0.
7.5	*Edge Case Behavior*	●	

Image Addressing and Filtering Implementation

The Altera SDK for OpenCL (AOCL) does not support image addressing and filtering. The AOCL does not support images.

Atomic Functions

Section 9 of the *OpenCL Specification version 1.0* describes a list of optional features that some OpenCL implementations might support. The Altera SDK for OpenCL (AOCL) supports atomic functions conditionally.

- Section 9.5: *Atomic Functions for 32-bit Integers*—The AOCL supports all 32-bit global and local memory atomic functions. The AOCL also supports 32-bit atomic functions described in Section 6.11.11 of the *OpenCL Specification version 1.1* and Section 6.12.11 of the *OpenCL Specification version 1.2*.
    - The AOCL does not support 64-bit atomic functions described in Section 9.7 of the *OpenCL Specification version 1.0*.

Attention: The use of atomic functions might lower the performance of your design. The operating frequency of the hardware might decrease further if you implement more than one type of atomic functions (for example, `atomic_add` and `atomic_sub`) in the kernel.

Embedded Profile Implementation

Section 10 of the *OpenCL Specification version 1.0* describes the OpenCL embedded profile. The Altera SDK for OpenCL (AOCL) conforms with the OpenCL embedded profile with clarifications and exceptions.

The table below summarizes the clarifications and exceptions to the OpenCL embedded profile:

Clause	Feature	Support Status	Notes
1	64-bit integers	●	
2	3D images	X	The AOCL does not support images.
3	Create 2D and 3D images with `image_channel_data_type` values	X	The AOCL does not support images.
4	Samplers	X	
5	Rounding modes	●	The default rounding mode for `CL_DEVICE_SINGLE_FP_CONFIG` is `CL_FP_ROUND_TO_NEAREST`.
6	Restrictions listed for single precision basic floating-point operations	X	
7	half type	X	This clause of the OpenCL Specification version 1.0 does not apply to the AOCL.
8	Error bounds listed for conversions from `CL_UNORM_INT8`, `CL_SNORM_INT8`, `CL_UNORM_INT16` and `CL_SNORM_INT16` to float	●	Refer to the table below for a list of allocation limits.

AOCL Allocation Limits

Item	Limit
Maximum number of contexts	Limited only by host memory size
Maximum number of queues	70 **Attention:** Each context uses two queues for system purposes.
Maximum number of program objects per context	20
Maximum number of even objects per context	Limited only by host memory size
Maximum number of dependencies between events within a context	1000
Maximum number of event dependencies per command	20

Item	Limit
Maximum number of concurrently running kernels	The total number of queues
Maximum number of enqueued kernels	1000
Maximum number of kernels per FPGA device	64
Maximum number of arguments per kernel	128
Maximum total size of kernel arguments	256 bytes per kernel

Document Revision History

Table A-4: Document Revision History of the Altera SDK for OpenCL Programming Guide Appendix A: Support Statuses of OpenCL Features

Date	Document Version	Changes
May 2015	15.0.0	• Listed the double precision floating-point functions that the Altera SDK for OpenCL supports preliminarily. • Added *OpenCL C Programming Language Restrictions for Pipes*.
December 2014	14.1.0	• In *AOCL Allocation Limits*, updated the maximum number of kernels per FPGA device from 32 to 64.
June 2014	14.0.0	• Updated the following AOCL allocation limits: • Maximum number of contexts • Maximum number of queues • Maximum number of even objects per context
December 2013	13.1.1	• Modified support status designations in *Appendix: Support Statuses of OpenCL Features*. • Removed information on OpenCL programming language restrictions from the section *OpenCL Programming Language Implementation*, and presented the information in a new section titled *OpenCL Programming Language Restrictions*.
November 2013	13.1.0	• Maintenance release.
June 2013	13.0 SP1.0	• Renamed *Optional Extensions* to *Atomic Functions*, and updated its content. • Removed *Platform Layer and Runtime Implementation*.
May 2013	13.0.1	• Maintenance release.

Date	Document Version	Changes
May 2013	13.0.0	• Added updated information on allocation limits and OpenCL language support.
November 2012	12.1.0	• Initial release.