Altera SDK for OpenCL

Programming Guide

Subscribe
Send Feedback

OCL002-14.0.0
2014.06.30

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

Contents

Altera SDK for OpenCL Programming Guide

1

2014.06.30

OCL002-14.0.0 Subscribe Send Feedback

About Altera SDK for OpenCL Programming Guide

The *Altera SDK for OpenCL Programming Guide* describes the contents and functionality of the Altera® Software Development Kit (SDK) for OpenCL™ (AOCL). The AOCL[1] is an OpenCL[2]-based heterogeneous parallel programming environment for Altera FPGAs.

This programming guide assumes that you are knowledgeable in OpenCL concepts and application programming interfaces (APIs), as described in the *OpenCL Specification version 1.0*. by the Khronos Group. This document also assumes that you have experience in creating OpenCL applications, and are familiar with the contents of the OpenCL Specification.

If you are using the AOCL or the RTE for a nonSoC device, this document assumes that you have read the *Altera SDK for OpenCL Getting Started Guide*, and have performed the following tasks:

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

Attention: If you have not performed the tasks described above, refer to the *Altera SDK 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™.

© 2014 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

If you are using the AOCL or the RTE for a Cyclone® V SoC Development Kit, this document assumes that you have read the *Altera SDK for OpenCL for Cyclone V SoC Getting Started Guide* or the *Altera RTE for OpenCL Getting Started Guide*, and 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
- For the RTE:
  - Download and install the Altera RTE for OpenCL
- Install and set up your Cyclone V SoC Development Kit
- Program your SoC with the hello_world example OpenCL application

Related Information

- **Altera SDK for OpenCL Getting Started Guide**
- **OpenCL Specification version 1.0**
  Refer to the *OpenCL Specification version 1.0* for detailed information on the OpenCL API and programming language.
- **OpenCL References Pages**
  Refer to the OpenCL Reference Pages for more information on the OpenCL Specification version 1.0.
- **Altera RTE for OpenCL Getting Started Guide**
- **Altera SDK for OpenCL Cyclone V SoC Getting Started Guide**

Contents of the AOCL

The Altera Software Development Kit (SDK) for OpenCL (AOCL) provides logic components, drivers, and AOCL-specific libraries and files.

Logic Components

- The *Altera Offline Compiler* (AOC) translates your OpenCL device code into a hardware configuration file that the system loads onto an Altera FPGA.
- The *AOCL Utility* includes a set of commands you can invoke to perform high-level tasks.
- The *host runtime* provides the OpenCL host platform application programming interface (API) and runtime API for your OpenCL host application.

  The host runtime consists libraries that provide OpenCL APIs, hardware abstractions, and helper libraries.

Drivers, Libraries and Files

The software installation process installs the software into a folder or directory referenced by the environment variable *ALTERAOCLSDKROOT*. The table below highlights some of the software contents.

Windows Folder	Linux Directory	Description
\bin	**/bin**	User commands in the AOCL. Include this folder or directory in your *PATH* environment variable.

Windows Folder	Linux Directory	Description
\board	**/board**	The Reference Platforms that come with the software. • For the s5_ref Reference Platform, the path is ***ALTERAOCLSDKROOT*/board/s5_ref** • For the Cyclone V SoC Development Kit Reference Platform, the path is ***ALTERAOCLSDKROOT*/board/c5soc**
\ip	**/ip**	Intellectual property (IP) core used to compile device kernels.
\host	**/host**	Files necessary for compiling and running your host application.
\host\include	**/host/include**	OpenCL Specification version 1.0 header files and software interface files necessary for compiling and linking your host program. The **CL** subfolder or subdirectory also includes the C++ header file **cl.hpp**. The file contains an OpenCL verison 1.1 C++ wrapper API. These C++ bindings enable a C++ host program to access the OpenCL runtime APIs using native C++ classes and methods. **Important:** The OpenCL version 1.1 C++ bindings are compatible with OpenCL Specification versions 1.0 and 1.1. Add this path to the **include** file search path in your development environment.
\host\windows64\lib	**/host/linux64/lib**	OpenCL host runtime libraries that provide the OpenCL platform and runtime APIs. These libraries are necessary for linking your host program. To run an OpenCL application on Linux, include this directory in the *LD_LIBRARY_PATH* environment variable.
\host\windows64\bin	**/host/linux64/bin**	Runtime commands and libraries necessary for running your host program, wherever applicable. Include this folder in your *PATH* environment variable. For Windows system, this folder only contains runtime libraries. For Linux system, this directory contains platform-specific binary for the `aocl` utility command.
\share	**/share**	Architecture-indepdendent support files.

Example OpenCL Applications

You can download example OpenCL applications from the **OpenCL Design Examples** page on the Altera website.

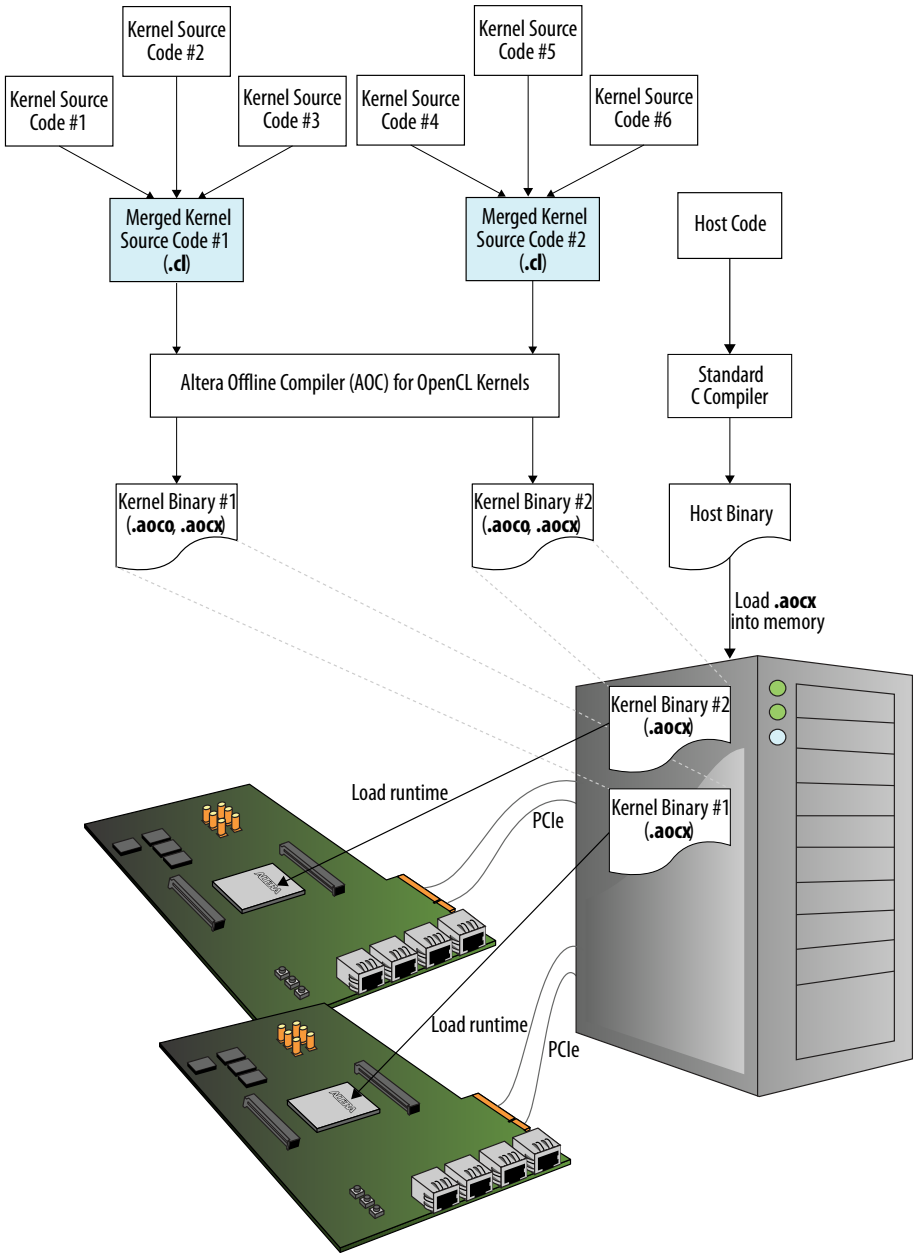
AOCL FPGA Programming Flow

The Altera Software Development Kit (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, and then the host-side C compiler compiles your host application and links the OpenCL kernels to it.

The following figure depicts the AOCL FPGA programming flow:

Figure 1-1: The AOCL FPGA Programming Flow



Important: 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 which 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 and then loads them into memory. The host runtime then calls these program objects from memory, and programs the target FPGA as required.

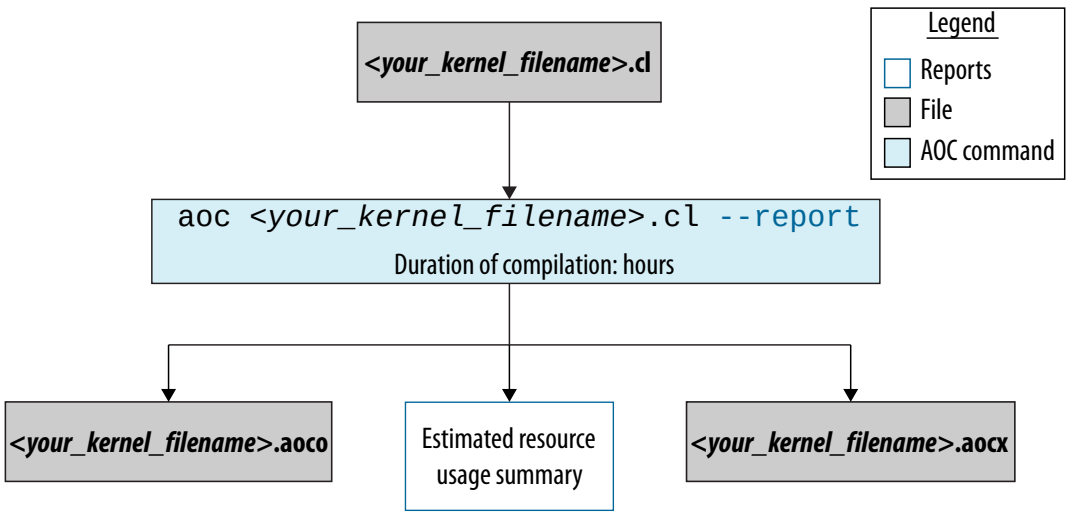
AOC Kernel Compilation Flows

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, as shown in the figure below. Choose this compilation option only if your OpenCL application requires minimal optimizations.

Figure 1-2: One-Step AOC Compilation Flow



Type the command `aoc <your_kernel_filename>.cl` to generate the hardware configuration file in a single step. During compilation, the AOC generates both the Altera Offline Compiler Object file (**.aoco**) and the Altera Offline Compiler Executable file (**.aocx**). The AOC also generates an estimated resource usage summary. To display the summary on-screen, add the `--report` option in your `aoc` command.

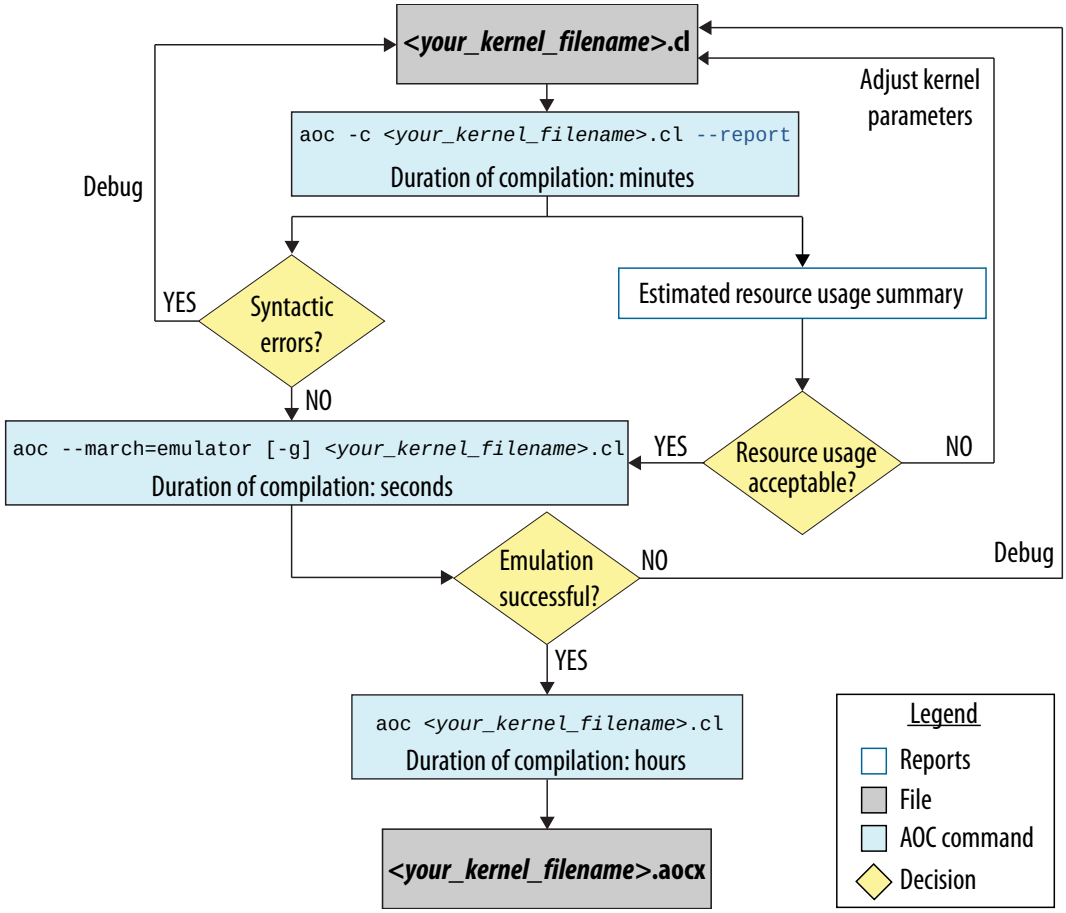
Important: The process of creating the **.aocx** file takes hours to complete.

Multistep Compilation for Complex Kernels

Choose the multistep compilation option if you want to implement optimizations to improve the performance of your OpenCL application.

The figure below illustrates the multistep compilation flow:

Figure 1-3: Multistep AOC Compilation Flow



To perform the first stage of compilation, include the `-c` option in the `aoc` command. The AOC generates the **.aoco** file and a <*your_kernel_filename*> folder or subdirectory. This compilation step only takes minutes to complete. If there are no syntactic errors in your code, you may emulate your kernel. Before you emulate your kernel, Altera recommends that you iterate on your design to ensure that its estimated resource usage fits the target device.

To create a **.aocx** file suitable for emulation on an x86-64 host system, include the `-march=emulator` option in your `aoc` command. For Linux systems, you can also add the `-g` option to enable symbolic debugging. After you finalize your kernel code, you can create the hardware configuration file.

Attention: Before proceeding to the final compilation stage, analyze and adjust your kernel code without building hardware to avoid long compilation times between iterations of your kernel code.

Compilation Reports

By default, the command prompt appears to signify the completion of the compilation. You can invoke the following AOC help options to generate reports that provide information on the progress of compilation and summarize the performance estimates of your kernel.

You can track the progress of kernel compilation by including the `-v` flag in the `aoc` command. The AOC notifies you of the compilation step it is currently performing.

The AOC calculates the estimated resource usage by default during compilation. You can review a summary of the estimated resource usage in the ***<your_kernel_filename>/<your_kernel_filename>*.log** file.

If you want to display the estimated resource usage summary on-screen, you can invoke the following command:

```
aoc -c <your_kernel_filename>.cl --report
```

Important: When you optimize your kernel code, generate the resource usage report for each iteration. The resource usage estimates can provide insight into the kind of optimizations you can implement to improve kernel performance.

Related Information

- **-c** on page 1-10
- **-v** on page 1-8
- **--report** on page 1-9

AOC Options

The Altera Software Development Kit (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.

AOC Help Options on page 1-8
You may include Altera Offline Compiler (AOC) help options in your `aoc` command to obtain information on the software, and on the compilation process.

AOC Overall Options on page 1-9
The Altera Offline Compiler (AOC) includes command line options that allow you to customize the compilation process such as compiling OpenCL kernels without hardware build, specifying names of output files, and defining macros.

AOC Modifiers on page 1-13
The Altera Offline Compiler (AOC) command line modifiers allow you to explore your FPGA board options and compile your kernels for a specific FPGA board.

AOC Optimization Controls on page 1-15
The Altera Offline Compiler (AOC) optimization controls direct the AOC to perform tasks such as partitioning memory, specifying logic utilization threshold, and modifying floating-point operations.

Related Information

OpenCL Design Examples
To practise using the `aoc` commands, download an example design from the OpenCL Design Examples page of the Altera website.

AOC Help Options

You may include Altera Offline Compiler (AOC) help options in your `aoc` command to obtain information on the software, and on the compilation process.

--version on page 1-8
To direct the Altera Offline Compiler (AOC) to print out the Altera Software Development Kit (SDK) for OpenCL (AOCL) version and then exits, invoke the `aoc --version` command.

-v on page 1-8
To direct the Altera Offline Compiler (AOC) to report on the progress of a compilation, include the `-v` option in your `aoc` command.

--report on page 1-9
To review the estimated resource usage summary on-screen, invoke the `aoc <your_kernel_filename>.cl --report` command.

--help or -h on page 1-9
To display a list of `aoc` command options, invoke the `aoc --help` or `aoc -h` command.

--version

To direct the Altera Offline Compiler (AOC) to print out the Altera Software Development Kit (SDK) for OpenCL (AOCL) version and then exits, invoke the `aoc --version` command.

Example output:

```
Altera SDK for OpenCL, 64-Bit Offline Compiler
Version 14.0 Build 200
Copyright (C) 2014 Altera Corporation
```

-v

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

For example, to direct the AOC to report on the progress of a full compilation, shown below, invoke the `aoc -v <your_kernel_filename>.cl` command.

```
aoc: Environment checks are completed successfully.
You are now compiling the full flow!!
aoc: Selected target board s5_ref
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.
```

You can use the `-v` option in conjunction with the `-c` option to receive progress report on an intermediate compilation step. For example, the AOC generates the following progress report when you invoke the `aoc -c -v <your_kernel_filename>.cl` command:

```
aoc: Environment checks are completed successfully.
aoc: Selected target board s5_ref
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"
```

For emulation, the AOC generates the following progress report when you invoke the `aoc -march=emulator -v <your_kernel_filename>.cl` command:

```
aoc: Environment checks are completed successfully.
You are now compiling the full flow!!
aoc: Selected target board s5_ref
aoc: Running OpenCL parser....ex
aoc: OpenCL parser completed successfully.
aoc: Compiling for Emulation ....
aoc: Emulator Compilation completed successfully.
Emulator flow is successful.
```

--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. You can review the estimated resource usage summary in the *<your_kernel_filename>***.log** file located in the *<your_kernel_filename>* folder or subdirectory. To review the estimated resource usage summary on-screen, invoke the `aoc <your_kernel_filename>.cl --report` command.

Below is an example of the AOC output when you invoke the `aoc -c <your_kernel_filename>.cl --report` command:

```
aoc: Selected target board s5_ref

+--------------------------------------------------------------------+
; Estimated Resource Usage Summary                                   ;
+----------------------------------------+---------------------------+
; Resource                               + Usage                     ;
+----------------------------------------+---------------------------+
; Logic utilization                      ; 18%                       ;
; Dedicated logic registers              ; 7%                        ;
; Memory blocks                          ; 16%                       ;
; DSP blocks                             ; 0%                        ;
+----------------------------------------+---------------------------;
```

--help or -h

To display a list of `aoc` command options, invoke the `aoc --help` or `aoc -h` command.

AOC Overall Options

The Altera Offline Compiler (AOC) includes command line options that allow you to customize the compilation process such as compiling OpenCL kernels without hardware build, specifying names of output files, and defining macros.

-c on page 1-10
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, invoke the `aoc -c <your_kernel_filename>.cl` command.

-o <filename> on page 1-10
To assign a specific filename to the output file, include the `-o <filename>` flag in your `aoc` command.

-O3 on page 1-11
To apply resource-driven optimizations to improve performance without violating fitting requirements, invoke the `aoc -O3 <your_kernel_filename>.cl` command.

-march=emulator on page 1-11
Include the `-march=emulator` option in your `aoc` command when you compile your OpenCL kernel for emulation.

-g on page 1-12
Include the `-g` option in your `aoc` command to add source references to compilation reports. It also adds debug data, and symbol and source references to your debug session.

-I <directory> on page 1-12
You can add <*directory*> to the list of directories that the Altera Offline Compiler (AOC) searches for header files during kernel compilation by including the `-I <directory>` flag in your `aoc` command.

-D <macro_name> or -D <macro_name=value> on page 1-13
You can define a preprocessor macro in your kernel source file by including the `-D <macro_name>` or `-D <macro_name=value>` flag in your `aoc` command.

-W on page 1-13
If you want to suppress all warning messages, include the `-W` flag when you invoke the `aoc` command.

-Werror on page 1-13
If you want to convert all warning messages into error messages, include the `-Werror` flag in your `aoc` command.

--big-endian on page 1-13
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), invoke the `aoc <your_kernel_filename>.cl --big-endian` command.

-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, 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 Altera Offline Compiler Object file (**.aoco**) in a matter of seconds to minutes. The AOC also creates a <*your_kernel_filename*> folder or subdirectory, which contains intermediate files that the Altera Software Development Kit (SDK) for OpenCL (AOCL) uses to build the hardware configuration file necessary for FPGA programming.

-o <*filename*>

To assign a specific filename to the output file, include the `-o <filename>` flag in your `aoc` command.

For example, if you compile an OpenCL kernel named **myKernel.cl**, specify the names of the output files in the following manner:

To specify the filename of the output **.aoco** file, type the following command:

```
aoc -c -o <your_object_filename>.aoco myKernel.cl
```

The `aoc` command you invoke to name the output Altera Offline Compiler Executable file (**.aocx**) depends on the Altera Offline Compiler (AOC) compilation flow you choose.

- If you implement the multistep compilation flow, specify the filename of your **.aocx** file by typing the following command:

  ```
  aoc -o <your_executable_filename>.aocx myKernel.aoco
  ```
- If you implement the one-step compilation flow, specify the filename of your **.aocx** file by typing the following command:

  ```
  aoc -o <your_executable_filename>.aocx myKernel.cl
  ```

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

Warning: Ensure that all filenames begin with alphanumeric characters. If the filename of your OpenCL application begins with a non-alphanumeric character, compilation will fail. For example, if you compile a kernel named **/&myKernel.cl** by typing `aoc /&myKernel.cl` at a command prompt, compilation fails with the following error message:

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

Warning: Ensure that all filenames end with alphanumeric characters. The AOC translates any non-alphanumeric character into an underscore ("_"). If you differentiate two filenames by ending them with different non-alphanumeric characters only (for example, **myKernel#.cl** and **myKernel&.cl**), the AOC translates both filenames to **myKernel_.cl**.

-O3

To apply resource-driven optimizations to improve performance without violating fitting requirements, invoke the `aoc -O3 <your_kernel_filename>.cl` command.

The estimated logic utilization should be less than or equal to 85%. You may override this logic utilization threshold by including the `--util <N>` option in the `aoc` command, where <*N*> is the maximum logic utilization for your kernel.

Related Information

- **Altera SDK for OpenCL Best Practices Guide**
  For more information, refer to the *Resource-Driven Optimization* section of the AOCL Best Practices Guide.
- **--util <N>** on page 1-16

-march=emulator

Include the `-march=emulator` option in your `aoc` command when you compile your OpenCL kernel for emulation.

Attention: Before you perform kernel emulation, ensure that you install a board package from your board vendor for your FPGA accelerator boards.

To create kernel programs executable on x86-64 host systems, invoke the `aoc -march=emulator <your_kernel_filename>.cl` command.

The emulator emulates the execution of your kernel on a target accelerator board. Ensure that the environment variable *AOCL_BOARD_PACKAGE_ROOT* points to the path to the Custom Platform from your board vendor. Alternatively, *AOCL_BOARD_PACKAGE_ROOT* should point to the path to a Reference Platform, available with the Altera Software Development Kit (SDK) for OpenCL (AOCL).

To specify a target board, you may include the `--board <board_name>` option in your `aoc` command (that is, `aoc -march=emulator --board <board_name> <your_kernel_filename>.cl`).

Related Information

- **--board <board_name>** on page 1-14
- **Emulate and Debug Your OpenCL Kernel** on page 1-60

-g

Include the `-g` option in your `aoc` command to add source references to compilation reports. It also adds debug data, and symbol and source references to your debug session.

The `-g` option serves two major purposes:

1. For Linux systems, when you include both the `-g` and the `-march=emulator` options in your `aoc` command (that is, `aoc -march=emulator -g <your_kernel_filename>.cl`), the Altera Offline Compiler AOC) enables symbolic debug support for the debugger. This allows you to pinpoint the origins of functional errors in your kernel source code.
2. When you compile a single work-item kernel, the AOC automatically generates an optimization report called **opt.rpt** in the ***<your_kernel_filename>*** subfolder or subdirectory. If you include the `-g` option in your `aoc` command (that is, `aoc -g <your_kernel_filename>.cl`), the AOC adds source information such as line numbers and variable names in the optimization report. The source information allows you to pinpoint the locations of loop-carried dependencies in your kernel source code.

Related Information

- **Debugging Your OpenCL Kernel (Linux)** on page 1-63
  For more information on the usage of the `-g` option in functional debugging, refer to the *Debugging Your OpenCL Kernel (Linux) section.*
- **Altera SDK for OpenCL Best Practices Guide**
  For more information on the usage of the `-g` option in single work-item kernel optimization, refer to the *Optimization Report* section of the AOCL Best Practices Guide.

-I *<directory>*

You can add *<directory>* to the list of directories that the Altera Offline Compiler (AOC) searches for header files during kernel compilation by including the `-I <directory>` flag in your `aoc` command.

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

-D <macro_name> or -D <macro_name=value>

You can define a preprocessor macro in your kernel source file by including the `-D <macro_name>` or `-D <macro_name=value>` flag in your `aoc` command.

To pass preprocessor macro definitions to the AOC, invoke the following command at the command line:

```
aoc –D <macro_name> <your_kernel_filename>.cl
```

or

```
aoc –D <macro_name=value> <your_kernel_filename>.cl
```

Related Information

Preprocessor Macros on page 1-39

-W

If you want to suppress all warning messages, include the `-W` flag when you invoke the `aoc` command.

To suppress all warning messages during compilation, invoke the `aoc -W <your_kernel_filename>.cl` command.

-Werror

If you want to convert all warning messages into error messages, include the `-Werror` flag in your `aoc` command.

To convert all warning messages into error messages during compilation, invoke the `aoc -Werror <your_kernel_filename>.cl` command.

--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), invoke the `aoc <your_kernel_filename>.cl --big-endian` command.

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

AOC Modifiers

The Altera Offline Compiler (AOC) command line modifiers allow you to explore your FPGA board options and compile your kernels for a specific FPGA board.

Remember: To view the list of available boards or to compile your kernel for a specific board, you must first set the environment variable *AOCL_BOARD_PACKAGE_ROOT* to point to the location of your Custom Platform.

Important: 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.

--list-boards on page 1-14

To print out a list of FPGA boards available in your board package, include the `--list-boards` flag in the `aoc` command.

--board <board_name> on page 1-14
To compile your OpenCL kernel for a specific FPGA board, invoke the `aoc --board <board_name> <your_kernel_filename>.cl` command.

--list-boards

To print out a list of FPGA boards available in your board package, include the `--list-boards` flag in the `aoc` command.

If you install a board package that includes more than one board type, when you 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_x*> is the board name you use in your `aoc` command to target a specific FPGA board.

--board *<board_name>*

To compile your OpenCL kernel for a specific FPGA board, invoke the `aoc --board <board_name> <your_kernel_filename>.cl` command.

If you want to compile the OpenCL application **myKernel.cl** for a specific FPGA accelerator board, first invoke the command `aoc --list-boards` to determine <*board_name*> from the list of available board types.

Assume the Altera Offline Compiler (AOC) outputs `FPGA_board_1` as the <*board_name*> of your target FPGA board. To compile **myKernel.cl** for `FPGA_board_1`, type `aoc --board FPGA_board_1 myKernel.cl` at the command prompt.

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

Tip: To identify readily 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.

For example, to target **myKernel.cl** to `FPGA_board_1` in the one-step compilation flow, invoke the command

```
aoc --board FPGA_board_1 myKernel.cl -o myKernel_FPGA_board_1.aocx
```

To target **myKernel.cl** to `FPGA_board_1` in the two-step compilation flow, invoke the commands

```
aoc -c --board FPGA_board_1 myKernel.cl -o myKernel_FPGA_board_1.aoco
```

and then

```
aoc --board FPGA_board_1 myKernel_FPGA_board_1.aoco -o
myKernel_FPGA_board_1.aocx
```

If you have an accelerator board consisting of two FPGAs, each FPGA device has an equivalent "board" name. If you have two FPGAs on a single accelerator board (for example, `board_fpga1` and

board_fpga2), and you want to target a **kernel_1.cl** to board_fpga1 and a **kernel_2.cl** to board_fpga2, invoke the following commands:

```
aoc --board board_fpga1 kernel_1.cl
aoc --board board_fpga2 kernel_2.cl
```

AOC Optimization Controls

The Altera Offline Compiler (AOC) optimization controls direct the AOC to perform tasks such as partitioning memory, specifying logic utilization threshold, and modifying floating-point operations.

--no-interleaving <global_memory_type> on page 1-15
You can include the `--no-interleaving <global_memory_type>` flag in the `aoc` command to manage buffer placement in global memory manually.

--const-cache-bytes <N> on page 1-16
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).

--util <N> on page 1-16
Include the `--util <N>` and the `-O3` flags in your `aoc` command to override the default logic utilization threshold.

--fp-relaxed on page 1-17
The `--fp-relaxed` flag directs the Altera Offline Compiler (AOC) to relax the order of arithmetic floating-point operations using a balanced tree hardware implementation.

--fpc on page 1-17
The `--fpc` flag directs the Altera Offline Compiler (AOC) to remove intermediary floating-point rounding operations and conversions whenever possible, and to carry additional bits to maintain precision.

--profile on page 1-17
To instrument the OpenCL kernel pipeline with performance counters, include the `--profile` option of the `aoc` command when you compile your kernel.

--no-interleaving *<global_memory_type>*

You can include the `--no-interleaving <global_memory_type>` flag in the `aoc` command to manage buffer placement in global memory manually. Manual partitioning of memory buffers overrides the default burst-interleaved configuration of global memory.

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.

If you invoke the `aoc <your_kernel_filename>.cl --no-interleaving default` command, the AOC disables burst-interleaving for the default global memory.

Note: 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.

Warning: The `--no-interleaving` flag requires a global memory type parameter. If you do not specify a memory type, the AOC issues an error message.

If you have a heterogeneous global memory system, and you want to disable burst-interleaving for one of the memory types (for example, DDR), invoke the following command:

```
aoc <your_kernel_filename>.cl --no-interleaving DDR
```

The AOC enables manual partitioning for the DDR memory bank, and configures the other memory bank in a burst-interleaved fashion.

If you have a heterogeneous memory system (for example, quad data rate (QDR) and DDR), and you want to partition manually both types of global memory buffers, invoke the following command:

```
aoc <your_kernel_filename>.cl --no-interleaving QDR --no-interleaving
DDR
```

Related Information

- **Partitioning Global Memory Accesses** on page 1-55
- **Kernel Attributes for Optimizing Memory Access Efficiency** on page 1-45

--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 following command:

```
aoc --const-cache-bytes <N> <your_kernel_filename>.cl
```

where <*N*> is the cache size in bytes.

Note: This argument has no effect if none of the kernels uses the `__constant` address space.

For example, to configure a 32 kB cache during compilation of the OpenCL kernel **myKernel.cl**, invoke the following command:

```
aoc --const-cache-bytes 32768 myKernel.cl
```

Related Information

Altera SDK for OpenCL Best Practices Guide
For more information on optimizing constant memory accesses, refer to the *Constant Cache Memory* section of the AOCL Best Practices Guide.

--util <N>

By default, the Altera Offline Compiler (AOC) performs resource-driven optimizations assuming that the logic utilization threshold is 85%. Include the `--util <N>` and the `-O3` flags in your `aoc` command to override the default logic utilization threshold.

To override the default logic utilization threshold, invoke the following command:

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

where <*N*> is the maximum percentage of FPGA hardware resources that the kernel uses.

Related Information

- **Altera SDK for OpenCL Best Practices Guide**
  For more information, refer to the *Resource-Driven Optimization* section of the AOCL Best Practices Guide.
- **-O3** on page 1-11

--fp-relaxed

The `--fp-relaxed` flag directs 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 following command:

```
aoc --fp-relaxed <your_kernel_filename>.cl
```

Related Information

Altera SDK for OpenCL Best Practices Guide
For more information, refer to the *Floating-Point Operations* section of the AOCL Best Practices Guide.

--fpc

The `--fpc` flag directs 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 to round 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 following command:

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

Related Information

Altera SDK for OpenCL Best Practices Guide
For more information, refer to the *Floating-Point Operations* section of the AOCL Best Practices Guide.

--profile

The Altera Software Development Kit (SDK) for OpenCL profiler relies on performance counters to gather kernel performance data. To instrument the OpenCL kernel pipeline with performance counters, include the `--profile` option of the `aoc` command when you compile your kernel.

Invoking the command `aoc --profile <your_kernel_filename>.cl` instruments the Verilog code in the ***<your_kernel_filename>*.aocx** Altera Offline Compiler Executable file with performance counters.

Attention: Instrumenting the Verilog code with performance counters increases hardware resource utilitzation (that is, increases FPGA area usage) and typically decreases performance.

Related Information

- **Altera SDK for OpenCL Best Practices Guide**
  For more information on the AOCL Profiler, refer to the *Profile Your Kernel to Identify Performance Bottlenecks* section of the AOCL Best Practices Guide.
- **Collecting Profile Data During Kernel Execution** on page 1-59

AOCL Utility

The Altera Software Development Kit (SDK) for OpenCL (AOCL) Utility provides you with tools and information to perform high-level tasks such as configuring the host application development flow.

General AOCL Utilities on page 1-18
Utility options that perform general tasks such as providing version and help information.

AOCL Utilities for Building Your Host Application on page 1-19
Utility options that provide components such as flags and libraries for compiling and linking your host application.

AOCL Utilities for Managing an FPGA Board on page 1-20
Utility options that direct the software to perform tasks such as installing and programming your FPGA board, and running diagnostic tests.

AOCL Profiler Utility on page 1-22
To view kernel performance data statistics that the Altera Software Development Kit (SDK) for OpenCL profiler collects, invoke the `aocl report <your_kernel_filename>.aocx profile.mon` command to launch the profiler GUI.

General AOCL Utilities

Utility options that perform general tasks such as providing version and help information.

version

To display the software version information, invoke the `aocl version` command.

Example output:

```
aocl 14.0.200 (Altera SDK for OpenCL, Version 14.0 Build 200,
Copyright (C) 2014 Altera Corporation)
```

help

To display a list of utility options, invoke the `aocl help` command.

To display the help content for a particular utility option, invoke the `aocl help <subcommand>` command.

For example, invoking the command `aocl help install` generates the following 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.
```

AOCL Utilities for Building Your Host Application

Utility options that provide components such as flags and libraries for compiling and linking your host application.

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

example-makefile or makefile

To display example makefile fragments for compiling and linking a host application, invoke the `aocl example-makefile` or `aocl makefile` command.

compile-config

To display the flags for compiling your host application, invoke the `aocl compile-config` command.

link-config or linkflags

To display the flags for linking your host application with the runtime libraries provided by the software, invoke the `aocl link-config` or `aocl linkflags` command.

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

ldflags

To display the linker flags necessary to link your host application to the host runtime libraries provided by the software, invoke the `aocl ldflags` command.

Attention: This utility does not list the libraries themselves.

ldlibs

To display the list of host runtime libraries provided by the software, invoke the `aocl ldlibs` command.

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 even 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.

To include in your host application the C header files that describe the OpenCL APIs, and to link your host application against the API libraries, perform the following tasks:

1. Type the command `aocl compile-config` at a command prompt.
   The software displays the path you must add to your C preprocessor. The path points to the folder or directory in which the OpenCL API header files reside.
   - The path is `-I%`*`ALTERAOCLSDKROOT`*`%\host\include` for Windows systems.
   - The path is `-I$`*`ALTERAOCLSDKROOT`*`/host/include` for Linux systems.

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

2. Invoke the `aocl link-config` command.
The output displays the link options for linking your host application against the appropriate OpenCL runtime libraries provided by the software.

Important: For Windows systems, you must add the `/MD` flag to link the host runtime libraries against the multi-threaded 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/<*platform*>/bin** folder in your library search path when you run your host application.

AOCL Utilities for Managing an FPGA Board

Utility options that direct the software to perform tasks such as installing and programming your FPGA board, and running diagnostic tests.

Identifying the Device Name of Your FPGA Board on page 1-20
When you query a list of accelerator boards, the software produces a list of installed devices on your machine in the order of their device names.

install on page 1-21
Invoke the `aocl install` utility command to install your FPGA accelerator boards and the operating system (OS) device driver into the current host system.

diagnose <device_name> on page 1-21
Invoke the `aocl diagnose <device_name>` utility command to run the board vendor's test program for your accelerator boards.

program <device_name> on page 1-21
Use the program utility to configure a new FPGA hardware image onto the accelerator board manually.

flash <device_name> on page 1-22
If supported, use the flash utility to initialize your FPGA with a specified startup configuration.

Related Information

Altera SDK for OpenCL Getting Started Guide
For an example on the usage of the AOCL `install` and `diagnose` utilities, refer to the *Installing an FPGA Board* section of the AOCL Getting Started Guide. For an example on the usage of the AOCL `flash` utility, refer to the *Programming the Flash Memory of an FPGA* section of the AOCL Getting Started Guide.

Identifying the Device Name of Your FPGA Board

Some Altera Software Development Kit (SDK) for OpenCL 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 software 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, perform the following tasks:

1. 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
```

install

The Custom Platform you acquire from your board vendor includes the device driver for your board. Invoke the `aocl install` utility command to install your FPGA accelerator boards and the operating system (OS) device driver into the current host system. The board-specific tools are referenced by the *AOCL_BOARD_PACKAGE_ROOT* environment variable.

Remember: To run `aocl install`, you must have administrator privileges.

Attention: For instructions on installing the Cyclone V SoC Development Kit, refer to the *Altera SDK for OpenCL Cyclone V SoC Getting Started Guide*.

diagnose *<device_name>*

Invoke the `aocl diagnose` *<device_name>* utility command to run the board vendor's test program for your accelerator boards.

To query a list of FPGA devices installed in your machine, invoke the `aocl diagnose` command.

To perform detailed diagnosis of a specific FPGA device, invoke the `aocl diagnose` *<device_name>* command, where <*device_name*> refers to the acl number (e.g. acl0 to acl15) that corresponds to your FPGA device.

To perform diagnostic tests on a Cyclone V SoC, invoke the `aocl diagnostic` command.

Important: Consult your board vendor's documentation for more information on using the diagnose utility to run diagnostic tests on multiple FPGA boards.

program *<device_name>*

Use the program utility to configure a new FPGA hardware image onto the accelerator board manually.

To program a specific FPGA device manually, invoke the `aocl program <device_name> <your_kernel_filename>.aocx` command, where <device_name> refers to the acl number (e.g. acl0 to acl15) that corresponds to your FPGA device, and **<your_kernel_filename>.aocx** is the Altera Offline Compiler Executable file (**.aocx**) that you use to program the hardware.

flash <device_name>

If supported, use the flash utility to initialize your FPGA with a specified startup configuration.

To program the flash memory of a specific FPGA device, invoke the command `aocl flash <device_name> <your_kernel_filename>.aocx`

where <device_name> refers to the acl number (e.g. acl0 to acl15) that corresponds to your FPGA device, and **<your_kernel_filename>.aocx** is the Altera Offline Compiler Executable file (**.aocx**) that you use to program the hardware.

Attention: For instructions on proramming the SD flash card of the Cyclone V SoC Development Kit, refer to the *Altera SDK for OpenCL Cyclone V SoC Getting Started Guide*.

AOCL Profiler Utility

To view kernel performance data statistics that the Altera Software Development Kit (SDK) for OpenCL profiler collects, invoke the `aocl report <your_kernel_filename>.aocx profile.mon` command to launch the profiler GUI.

If you use the `--profile` option of the `aoc` command to compile an OpenCL kernel, the Altera Offline Compiler (AOC) instruments the resulting Altera Offline Compiler Executable file (**.aocx**) with performance counters. When you execute the hardware configuration file on the FPGA, the host collects receives the performance data and stores it in a file called **profile.mon** in the current working directory.

Related Information

- **Altera SDK for OpenCL Best Practices Guide**
  For more information on the AOCL Profiler, refer to the *Profile Your Kernel to Identify Performance Bottlenecks* section of the AOCL Best Practices Guide.
- **--profile** on page 1-17

Kernel Programming Considerations

Altera offers guidelines on how to structure your kernel code. To increase efficiency, implement these programming considerations when you create a kernel or modify a kernel written originally to target another architecture.

AOCL Channels Extension on page 1-23
The Altera Software Development Kit (SDK) for OpenCL (AOCL) channels extension provides a mechanism for passing data to kernels and synchronizing kernels with high efficiency and low latency.

Preprocessor Macros on page 1-39
The Altera Offline Compiler (AOC) supports preprocessor macros that allow you to pass macro definitions and compile code on a conditional basis.

__constant Address Space Qualifiers on page 1-40
There are several limitations and workarounds you must consider when you include __constant address space qualifiers in your kernel.

Pragmas and Attributes on page 1-41
You can increase the data processing efficiency of your OpenCL kernel by specifying pragmas and attributes.

Use Structure Data Types as Arguments in OpenCL Kernels on page 1-47
Convert each structure parameter (struct) to a pointer that points to a structure.

Register Inference on page 1-49
When you declare data in qualified or unqualified private memory address space (that is, __private), the Altera Offline Compiler (AOC) attempts to promote the data into registers.

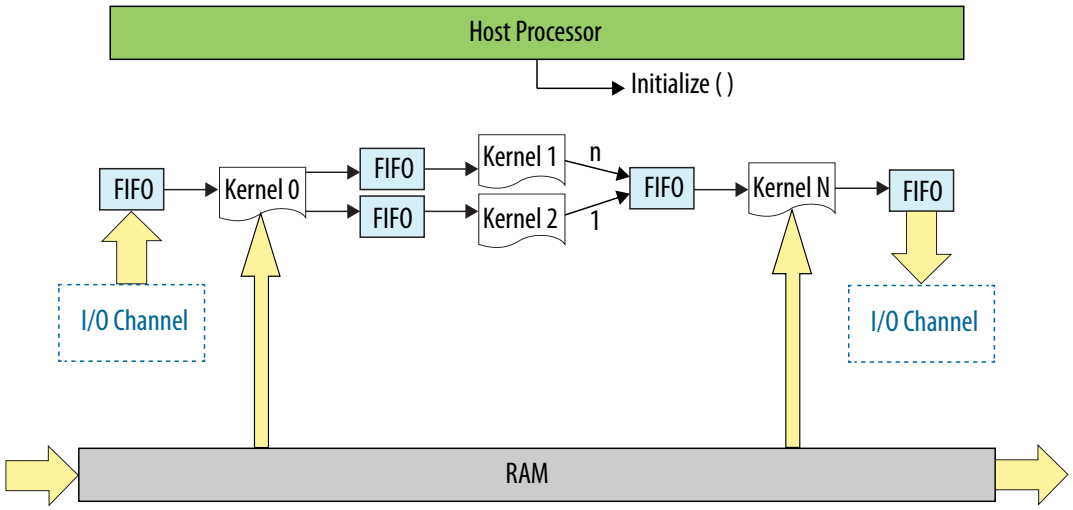
AOCL Channels Extension

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

The 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



Channels API on page 1-24
To implement the Altera Software Development Kit (SDK) for OpenCL (AOCL) channels extension, you have to modify your OpenCL kernels to include channels-specific pragma and application programming interface (API) calls.

Channel Data Behavior on page 1-28
Data written to a channel remains in a channel as long as the kernel program remains loaded on the FPGA device.

Channels API

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

The OPENCL EXTENSION Pragma

To enable the Altera Software Development Kit (SDK) for OpenCL (AOCL) channels extension, you must declare the `OPENCL EXTENSION` pragma at the beginning of your kernel source file.

Declare the `OPENCL EXTENSION` pragma as follows:

```
#pragma OPENCL EXTENSION cl_altera_channels : enable
```

Channel Handle Declaration

You can define the connectivity between kernels, or between kernels and I/O, using the `channel <type> <variable_name>` channel variable.

You must declare the channel variable as a file scope variable in the kernel source file (**.cl**) using the `channel` keyword. For example, to declare a channel of type `int`, include the following declaration:

```
channel int c;
```

Channel declarations are unique within a given OpenCL kernel program. In addition, 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.

The Altera Software Development Kit (SDK) for OpenCL (AOCL) channels extension supports simultaneous channel accesses by multiple variables declared in a structure.

Declare a struct for a channel in the following manner:

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

channel type_t foo;
```

Important: The channel type must have a data size of 1024 bits or less. If you send pointer values through a channel, know that the OpenCL memory model does not require kernels to have access to the same memory regions.

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

Blocking Channel Writes (write_channel_altera)

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

The `write_channel_altera` function signature is as follows:

```
void write_channel_altera (channel <type> channel_id, const <type> data);
```

This function signature takes in two parameters: `channel_id` and `data`. The `channel_id` parameter identifies the buffer to which the channel connects, and it must match the `channel_id` of the corresponding read channel (`read_channel_altera`). The `data` parameter is the data the channel write operation writes to the channel. Data <*type*> must match the <*type*> of the `channel_id`. The <*type*> of a channel defines its data width, which cannot be a constant.

The following code example 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 mnid that if the channel is full (that is, if the FIFO buffer is full of data), your kernel will stall. Use the Altera Software Development Kit (SDK) for OpenCL profiler to check for channel stalls.

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.

Important: Follow the OpenCL conversion rules to ensure that data the kernel writes to a channel is convertible to <*type*>.

Nonblocking Write Extensions

You can perform nonblocking writes to facilitate applications where data write operations might not occur.

The signature of nonblocking writes is similar to blocking writes; however, it returns an boolean value that indicates whether data is written to the channel:

```
bool write_channel_nb_altera(channel <type> channel_id, const <type> data);
```

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

The following producer code segment accomplishes this work distribution:

```
#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);
    }
}
```

Blocking Channel Reads (read_channel_altera)

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

The `read_channel_altera` function signature is as follows:

```
<type> read_channel_altera(channel <type> channel_id);
```

This function takes in two parameters: `channel_id` and `data`. The `channel_id` parameter identifies the buffer to which the channel connects, and it must match the `channel_id` of the corresponding write channel, as specified in Channel Handle declaration (that is, `channel <type> <variable_name>`).

The following code example 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.

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.

Important: Ensure that the variable the kernel assigns to read the channel data is convertible from *<type>*.

Nonblocking Read Extensions

You can 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:

```
<type> read_channel_nb_altera(channel <type> channel_id, bool * valid);
```

The following code segment demonstrates the usage of a nonblocking read API call:

```
#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);
        }
    }
}
```

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:

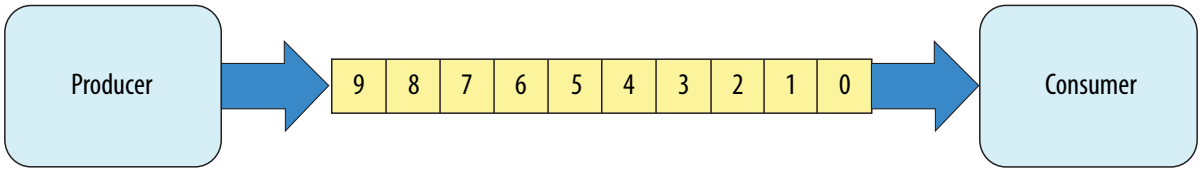
```
#pragma OPENCL EXTENSION cl_altera_channels : enable
channel int c0;

__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 writes the elements to the channel:

Figure 1-5: Channel Data FIFO Ordering



The kernel `producer` writes 10 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 occuring, 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.

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 Feedforward 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 Software Development Kit (SDK) for OpenCL (AOCL) channels extension does not support dynamic indexing into arrays of channel IDs.

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.

Unbuffered versus Buffered Channels

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.

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. Buffered channels decouples the operation of concurrent threads executing in different kernels. The main purpose of buffered channels is to help prevent kernel stalls when there is a temporary imbalance in the execution of read and write operations to a channel.

Related Information

Attributes for Channels on page 1-46

Buffered Channels

Buffered channels decouples the operation of concurrent threads executing in different kernels. Buffered channels help prevent kernel stalls when there is a temporary imbalance in the execution of read and write operations to a channel.

If you expect any temporary mismatch between the consumption rate and the production rate to the channel, set the buffer size. Consider the following example:

```
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;
    }
}
```

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 there exists any condition where the producer kernel is writing data to the channel when the consumer kernel is not reading from it.

I/O Channels

You may 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.

Below is a code example on the usage of the `io` channel attribute:

```
channel int io_chan __attribute__((io("input_stream")));

__kernel void stream_to_memory( __global int * out_data, int N )
{
    // read 32-bit data from stream and write the data to global memory
    for(int i = 0; i < N; i++)
    {
        out_data[i] = read_channel_altera(io_chan);
    }
}
```

Tip: To determine which input or output features are available on your FPGA board, consult the **board_spec.xml** file in your Custom Platform.

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

Caution: Implicit data dependencies might exist for channels that connect directly to the board 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 types of dependencies.

Caution: 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.

Related Information

Attributes for Channels on page 1-46

Channel Interaction Control Using Memory Fence or Barrier Functions

To enforce the ordering of channel calls, you can introduce memory fence or barrier functions in your kernel programs. These functions enforce the order of channel calls by controlling memory accesses.

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, a deadlock might occur if you construct a `producer` kernel and a `consumer` kernel in the following manner:

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

Channels `c0` and `c1` are unbuffered channels. Because the schedule of the read channel calls might place the read operation from `c1` before `c0`, a deadlock might occur.

To prevent deadlocks from occurring, you must insert memory fence functions (`mem_fence`) to enforce the ordering of channel calls. The following example demonstrates that the `mem_fence` call with the channel flag forces the sequential ordering of the write and read channel calls in the `producer` and `consumer` kernels:

```
__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);
    }
}
```

External Memory Consistency across Kernels

According to the OpenCL Specification version 1.0, memory behavior is undefined unless a kernel completes execution. In other words, 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 do so, you must define memory consistency across kernels with respect to memory fences, as shown in the code example below:

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

Without a memory fence function (`mem_fence`), 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.

A `mem_fence` is necessary to create a control flow dependence between the channel synchronization calls before the the fence and after the fence. 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`.

If you want 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, as shown below:

```
__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]`.

Multiple Work-Item Ordering

The OpenCL specification does not define a work-item ordering. The Altera Software Development Kit (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 key 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 to ensure 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. The primary case where the AOCL fails to provide deterministic ordering is 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

Work-item serial execution refers to an ordered execution behavior where work-item sequential IDs determine the order in which they execute 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 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 an sequential order. Defining this order ensures that the system is verifiable with external models.

Channel Execution in Multiple Work-Items Loop

When channels exist in the body of a multiple work-items loop, 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)]);
}
```

Use Models

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

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

Feed-Forward Design Model

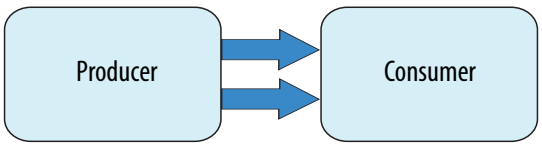
Implement the feed-forward design model to send data from one kernel to the next without creating any cycles between them. Consider the following code example:

```
__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` kernels 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

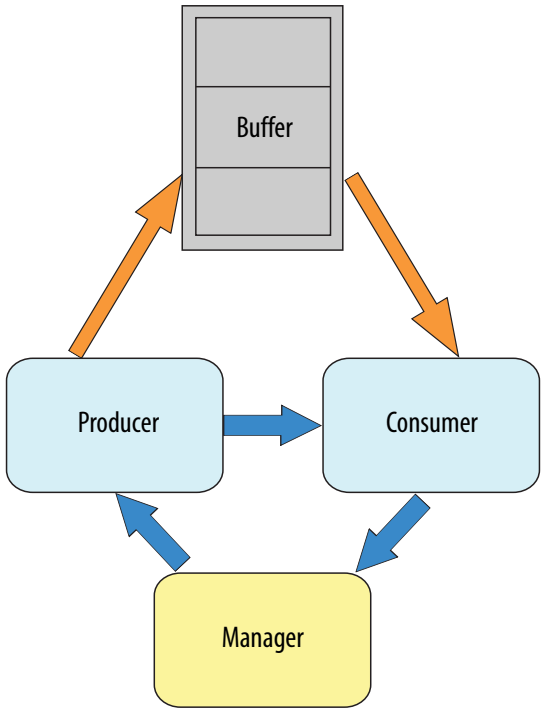


Buffer Management

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:

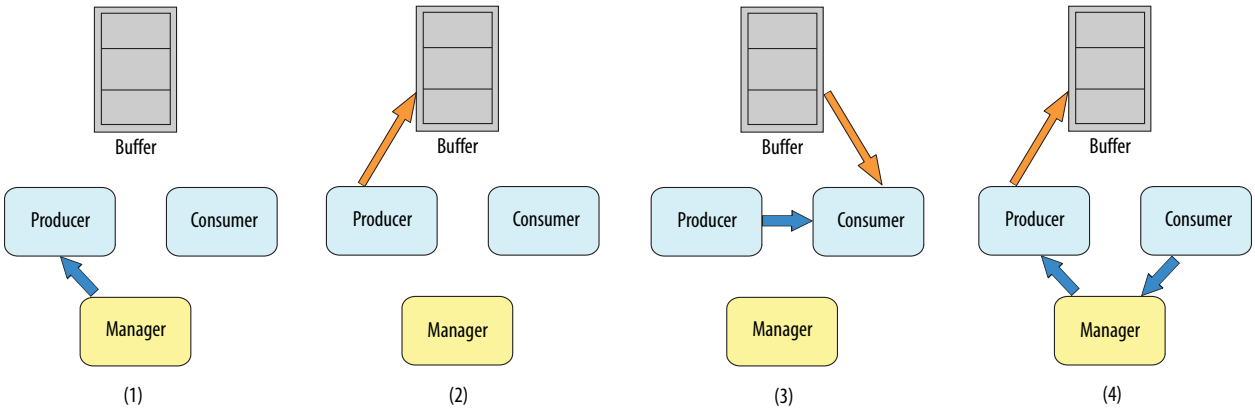
Figure 1-7: Feed-Forward Design Model with Buffer Management



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 the `producer` to use.
2. After the `manager` kernel allocates the memory region , the `producer` writes data to that region of the ping-pong buffer.
3. After the `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` then reads data from that region of the ping-pong buffer.

   Note: When the `consumer` is performing the read operation, the `producer` can write to other free memory locations for processing because of the concurrent execution of the `producer`, `consumer`, and `manager` kernels.
4. After the `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 the `producer` kernel to use.

Implementation of Buffer Management for AOCL Kernels

To ensure the Altera Software Development Kit (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` commits data to memory. In other words, the channel write operation cannot occur until the `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.

Preprocessor Macros

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

To pass preprocessor macro definitions to the AOC, include the `-D <macro_name>` or `-D <macro_name=value>` flag in your `aoc` command.

For example, if you want to control the amount of loop unrolling, you can define a preprocessor macro for the unrolling factor, as shown in the example below:

```
#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;
}
```

For the kernel `sum`, shown above, invoke the following command to override the existing value for the `UNROLL_FACTOR` macro and set it to 4:

```
aoc –D UNROLL_FACTOR=4 sum.cl
```

Invoking this command is equivalent to replacing the line `#define UNROLL_FACTOR 1` with `#define UNROLL_FACTOR 4` in the `sum` kernel source code.

You can use preprocessor macros to control how the AOC optimizes your kernel without modifying your kernel source code. 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
```

Attention: 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.

Conditional Compilation Based on Preprocessor Macros

You can compile your kernel with conditional parameters and features by defining preprocessor macros.

For example, compiling your kernel with the `--board <board_name>` flag sets the preprocessor macro `AOCL_BOARD_<board_name>` to 1. If you target your kernel to an FPGA board named FPGA_board_1, you can include device-specific parameters in your kernel code in the following manner:

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

Another example is the Altera predefined preprocessor macro `ALTERA_CL`, which you can use to introduce AOC-specific compiler features and optimizations, as shown below:

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

Related Information

- **--board <board_name>** on page 1-14
- **Kernel Attributes for Specifying Work-Group Sizes** on page 1-42

__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, which 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.

Pragmas and Attributes

You can increase the data processing efficiency of your OpenCL kernel by specifying pragmas and attributes. These pragmas and attributes direct the Altera Offline Compiler (AOC) to process your work-items and work-groups in a way that increases throughput. The AOC conducts static performance estimations, and automatically selects operating conditions that do not degrade performance.

OpenCL Kernel Pragmas on page 1-42

Kernel Attributes for Specifying Work-Group Sizes on page 1-42

Kernel Attributes for Optimizing Data Processing Efficiency on page 1-43

Kernel Attributes for Optimizing Memory Access Efficiency on page 1-45

Attributes for Channels on page 1-46

Related Information

Altera SDK for OpenCL Best Practices Guide

For more information on kernel pragmas and attributes, refer to the *Strategies for Improving NDRange Kernel Data Processing Efficiency* and the *Strategies for Improving Memory Access Efficiency* sections of the AOCL Best Practices Guide. For more information on heterogeneous memory support, refer to the *Heterogeneous Memory Buffers* section of the AOCL Best Practices Guide.

OpenCL Kernel Pragmas

- `unroll`—instructs the AOC to unroll a loop.

  When you include the `#pragma unroll <N>` directive, 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];
}
```

  The AOC might unroll simple loops even if they are not annotated by a pragma. If you do not specify a value for <*N*>, 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.

  Attention: Provide an unroll factor whenever possible.

Kernel Attributes for Specifying Work-Group Sizes

- `max_work_group_size`—specifies the maximum number of work-items that the AOC can allocate to a work-group in a kernel.
- `reqd_work_group_size`—specifies the required work-group size .

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

Important: 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.

Related Information

Altera SDK for OpenCL Best Practices Guide

For more information, refer to the *Specifying a Maximum Work-Group Size or a Required Work-Group Size* section of the AOCL Best Practices Guide.

Kernel Attributes for Optimizing Data Processing Efficiency

- `num_compute_units(<N>)`—specifies <*N*> number of compute units the AOC instantiates to process the kernel.

  The AOC distributes work-groups across the specified number of compute units.

  For example, the code fragment below directs the AOC to instantiate two compute units in a kernel by specifying the `num_compute_units` attribute:

  ```
  __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 increased number of compute units achieves higher throughput at the expense of global memory bandwidth contention among compute units.
- `num_simd_work_items(<N>)`—specifies <*N*> number of work-items within a work-group the AOC executes in a single instruction multiple data (SIMD) manner.

  The AOC replicates the kernel datapath according to the value you specify for this attribute whenever possible.

  Important: You must 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.

  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];
  }
  ```

- `packed`—instructs the AOC to disable automatic insertion of data structure padding between data members of a `struct`.

  For example, the code fragment below directs the AOC to disable the insertion of data structure padding:

```
__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
   }
}
```

- `aligned(<N>)`—specifies <*N*> amount of data structure padding the AOC applies between data members of a `struct`.

  For example, the code fragment below directs the AOC to insert two data structure padding:

```
__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
   }
}
```

Kernel Attributes for Optimizing Memory Access Efficiency

- `local_mem_size(<N>)`—specifies a pointer size in <*N*> bytes, other than the default size of 16 kB, to optimize the local memory hardware footprint (size).

  For example, you can use the `local_mem_size` attribute in your pointer declaration in the following manner:

  ```
  __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`.
- `buffer_location("<memory_type>")`—defines the global memory type in which you can allocate a buffer.

  For example, if your FPGA board includes DDR and QDR memory types, you can define the memory types as follows:

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

  Important: 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, you can access the **board_spec.xml** file in your Custom Platform, and search for the memory type that is defined first or has the attribute `default=1` assigned to it.

Attributes for Channels

- `depth(<N>)`—specifies the minimum depth of a buffered channel, where <*N*> is the number of data values.

  The following example includes the `depth` attribute to the channels program scope variable to declare a buffered channel:

  ```
  #pragma OPENCL EXTENSION cl_altera_channels : enable

  channel long chan __attribute__((depth(10)));

  kernel void kernel_write_channel(global long * src)
  {
      for(i=0; i < N; i++)
      {
          write_channel_altera(chan, src[i]);
      }
  }
  ```

  In this example, the write operation can write 10 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.
- `io("<chan_id>")`—specifies the I/O feature of an accelerator board with which a channel interface, where <*chan_id*> is the name of the I/O interface listed in the **board_spec.xml** file of your Custom Platform.

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

  ```
  <channels>
      <interface
          name = "ethernet"
          port = "dataout"
          type = "streamsource"
          width = "32"
          chan_id = "enet"
      />
      <interface
          name = "pcie"
          port = "tx"
          type = "streamsink"
          width = "32"
          chan_id = "pcie"
      />
  </channels>
  ```

  Important: The `width` attribute of an `interface` 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.

  To utilize the I/O channel in your kernel, include the `io` attribute for each feature, as shown below:

  ```
  channel uint pkt __attribute__((io("enet")));
  channel float data __attribute__((io("pcie")));
  ```

  Important: The `io` kernel attribute names must match those of the I/O channels (`chan_id`) specified in the **board_spec.xml** file.

Use 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:	Rewrite your code to resemble the following syntax:

`struct Context`
`{`
`  float param1;`
`  float param2;`
`  int param3;`
`  uint param4;`
`};`
 	
`__kernel void algorithm(__global float * A,`
`struct Context c)`
`{`
`  if ( c.param3 )`
`  {`
`  // statements`
`  }`
`}`
	`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.

Match Data Layouts of Host and Kernel Structure Data Types

If you use structure data types (`structs`) as arguments in OpenCL kernels, you must 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 avaiable 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`.

You must align the structures and align the `struct` data members between the host and kernel applications.

Attention: You must manage the alignments carefully because of the variability among different host compilers.

For example, if you have OpenCL data types (for example, `float4`) in the struct, the alignments of these data items must satisfy the OpenCL specification (for example, 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 staisfy the alignment requirements for each data member.

   a. In your OpenCL kernel code, you may specify data packing (i.e. 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 Software Development Kit for OpenCL (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, declare the `struct` in the following manner:

```
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;
};
```

   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.

      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 1-59

Register Inference

When you declare data in qualified or unqualified private memory address space (that is, `__private`), the Altera Offline Compiler (AOC) attempts to promote the data into registers. The promotion is beneficial for data access that occurs in a single cycle (for example, feedback in a single work-item loop).

The AOC promotes arrays either as single values or in a piecewise fashion. Piecewise promotion results in very efficient hardware; however, the AOC must be able to determine data accesses statically. To facilitate this, hardcode the access points into the array. In addition, you can also facilitate register promotion by unrolling loops that access the array.

If array accesses are not statically inferable, the AOC might promote the array into 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 statically inferable 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 promotes `array[i]` into block RAMs. If the size of `array[i]` is greater than 64 bytes, or if the kernel has multiple work-items, the AOC promotes 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 promotes the array to block RAMs and instantiates specialized hardware for each load or store operation.

Shift Register Inference

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, do not initialize any element. Uninitialized elements have a value of 0.

Attention: If not all accesses to a large array are statically inferable, they force the AOC to create inefficient hardware. If these accesses are necessary, use `__local` memory instead of `__private` memory.

Host Programming Considerations

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

Channels and Multiple Command Queues on page 1-51
You must instantiate a separate command for each kernel that you wish to run concurrently.

Host Binary Requirement on page 1-51
When compiling the host application, you must target one of these architectures: x86-64 (64-bit), IBM POWER (64-bit), or ARM® 32-bit ARMV7-A for Cyclone V SoC.

Host Machine Memory Requirements on page 1-51
The machine that runs the host application must have enough host memory to support several components simultaneously.

FPGA Programming on page 1-51
The Altera Offline Compiler (AOC) is an offline compiler that compiles kernels independent of the host application. To load the kernels into the OpenCL runtime, use the `clCreateProgramWithBinary` function in your host application.

Multiple Host Threads on page 1-55
The Altera Software Development Kit (SDK) for OpenCL (AOCL) host library is not thread-safe.

Partitioning Global Memory Accesses on page 1-55
Manual partitioning of global memory buffers allows you to control memory accesses across buffers to maximize the memory bandwidth.

Shared Memory Accesses for OpenCL Kernels Running on SoCs on page 1-57
Altera recommends that OpenCL kernels that run on Altera SoCs access shared memory instead of the FPGA DDR memory.

Out-of-Order Command Queues on page 1-59
The OpenCL host runtime command queues do not support out-of-order command execution.

Modifying Host Program for Structure Parameter Conversion on page 1-59
If you convert any structure parameters to pointers-to-constant structures in your OpenCL kernel, you must modify your host application accordingly.

Collecting Profile Data During Kernel Execution on page 1-59
In cases where kernel execution finishes after the host application completes, you can query the FPGA explicitly to collect profile data during kernel execution.

Channels and Multiple Command Queues

The Altera Software Development Kit (SDK) for OpenCL (AOCL) channels extension allows multiple kernels to execute in parallel. However, the AOCL channels facilitate this concurrent behavior only when `cl_command_queue` objects are in order. You must instantiate a separate command for each kernel that you wish to run concurrently.

Host Binary Requirement

When compiling the host application, you must target one of these architectures: x86-64 (64-bit), IBM POWER (64-bit), or ARM® 32-bit ARMV7-A for Cyclone V SoC. The Altera Software Development Kit (SDK) for OpenCL (AOCL) host runtime does not support x86-32 (32-bit) binaries.

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.

FPGA Programming

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

The following host code demonstrates the usage of the `clCreateProgramWithBinary` function 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);
```

After the AOC builds the Altera Offline Compiler Executable file (**.aocx**), you may use the file to create the OpenCL device program. You can use either `clCreateKernelsInProgram` or `clCreateKernel` to create kernel objects. Instead of `clCreateProgramWithSource`, the `clCreateProgramWithBinary` function uses the **.aocx** file to create `cl_program` objects for the target FPGA. You can load multiple FPGA programs into memory. The host runtime then reprograms the FPGA as required to execute the scheduled kernels via the `clEnqueueNDRangeKernel` and `clEnqueueTask` application programming interface (API) calls.

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.

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 to 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`).

Querying Device Information

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

1. 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, and <*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 FPGAs, each device is treated as a "board" and is given a unique name.

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 `createMultiDeviceProgram` 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);
```

Multiple Host Threads

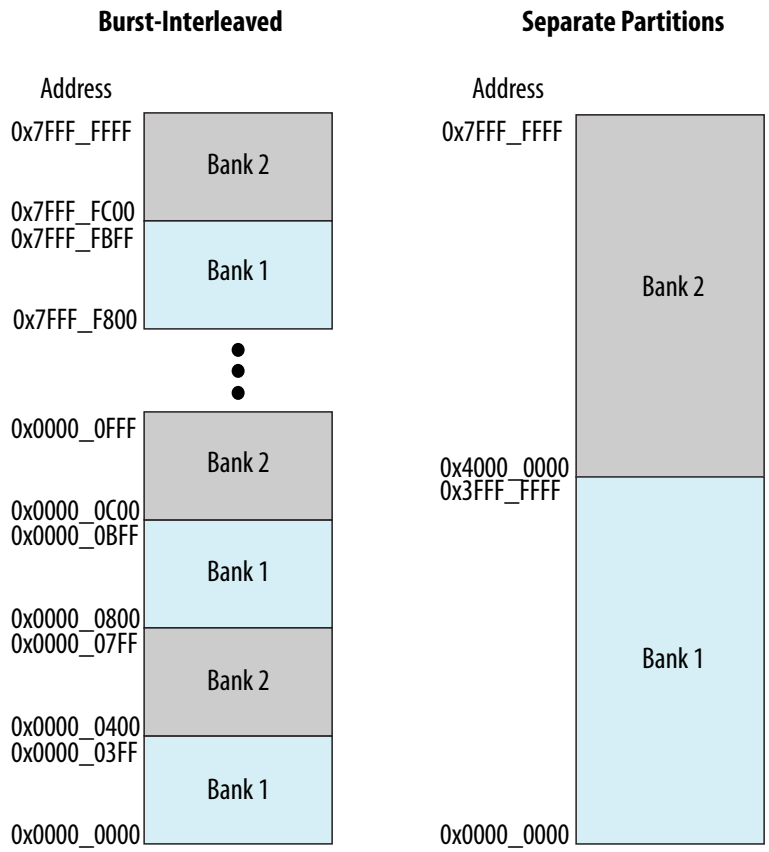
The Altera Software Development Kit (SDK) for OpenCL (AOCL) host library is not thread-safe.

Partitioning Global Memory Accesses

Manual partitioning of global memory buffers allows you to control memory accesses across buffers to maximize the memory bandwidth. Before you perform the partitioning, first you have to disable burst-interleaving by including the `--no-interleaving <global memory_type>` flag of the `aoc` command.

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

1. At a command prompt, type `aoc --no-interleaving <global_memory_type> <your_kernel_filename>.cl` to compile your OpenCL kernels and configure the memory bank(s) of the specified memory type as separate address spaces.
   a. If you want to partition more than one memory type manually, add a `--no-interleaving <global_memory_type>` flag for each memory type.

      To determine the name of each type of global memory (that is, <*global_memory_type*>), consult the **board_spec.xml** file in your Custom Platform.

      Refer to your board vendor's Custom Platform documentation for the location of the **board_spec.xml** file.
   b. If you want to disable interleaving for the default memory type, invoke the `aoc --no-interleaving default <your_kernel_filename>.cl` command.
2. When you create an OpenCL buffer in your host program, 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).

   By default, the host allocates buffers into the main memory when you load kernels into the OpenCL runtime via the `clCreateProgramWithBinary` function. As a result, upon kernel invocation, the host relocates heterogenous memory buffers that are bound to kernel arguments to the main memory

automatically. To avoid the initial allocation of heterogeneous memory buffers in the main memory, include the `CL_MEM_HETEROGENEOUS_ALTERA` flag when you use the `clCreateBuffer` function, as shown below:

Caution: Do not pass a buffer as kernel arguments that associate it with multiple memory technologies.

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

```
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 non-default 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.

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`.

Related Information

- **Altera SDK for OpenCL Best Practices Guide**
  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 AOCL Best Practices Guide.
- **--no-interleaving <global_memory_type>** on page 1-15
- **Kernel Attributes for Optimizing Memory Access Efficiency** on page 1-45

Shared Memory Accesses for OpenCL Kernels Running on SoCs

Altera recommends that OpenCL kernels that run on Altera SoCs access shared memory instead of the FPGA DDR memory. FPGA DDR memory is accesible 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.

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.

The following host code example demonstrates how to allocate and access shared memory:

```
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
```

Note: 1. Mark the shared buffers between kernels as volatile. This ensures that when one kernel modifies a buffer, the change 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 shared memory. Also, the `CL_MEM_USE_HOST_PTR` flag does not work with shared memory. In this context, shared memory is physical shared memory, not virual shared memory. The FPGA accesses the same DDR controller as the host system, but the FPGA cannot access the virtual page tables that the host system uses to translate virtual addresses to physical addresses.
   a. In DDR memory, shared memory must be physically contiguous. You can use `malloc` or `new` to access memory that is virtually contiguous, not physically. The FPGA cannot consume virtually contiguous memory without a scatter-gather direct memory access (SG-DMA) controller core.
4. CPU caching is disabled for the shared memory.
5. 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 `clCreate-Buffer(..., CL_MEM_READ_WRITE, ...)` allocates to shared memory, similar to using the `CL_MEM_ALLOC_HOST_PTR` flag.
6. 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 OS Kernel Source* and the *Altera OpenCL SoC Driver* sections of the *Altera Cyclone V SoC Development Kit Reference Platform User Guide*.

   Attention: The `CONFIG_CMA_SIZE_MBYTES` kernel configuration option controls 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`.

Alternatively, 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. The kernel transfers memory from HPS DDR memory to the FPGA DDR memory.

```
__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: You must still allocate the `src` pointer in the HPS DDR as shared memory using the `CL_MEM_ALLOC_HOST_PTR` flag, as described in the host code above the Note.

Related Information

Altera Cyclone V SoC Development Kit Reference Platform User Guide

Out-of-Order Command Queues

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

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

- **Use Structure Data Types as Arguments in OpenCL Kernels** on page 1-47
- **Match Data Layouts of Host and Kernel Structure Data Types** on page 1-47

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 compile your OpenCL kernel with the `aoc --profile <your_kernel_filename>.cl` command, the **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 does not complete before 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.

1. 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.

Related Information

Altera SDK for OpenCL Best Practices Guide

For more information on the AOCL Profiler, refer to the *Profile Your Kernel to Identify Performance Bottlenecks* section of the AOCL Best Practices Guide.

Emulate and Debug Your OpenCL Kernel

Use the Altera Software Development Kit (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 funcationality 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.

Prerequisites for Emulation

Prior to kernel emulation, ensure that you set all environment variables and perform all necessary kernel modifications.

1. To emulate your kernels on Windows systems, you need the Microsoft linker and additional compilation time libraries.
   a. Ensure that your *PATH* environment variable setting includes the path to the Microsoft **LINK.EXE** file.

      The **LINK.EXE** file is available with Microsoft Visual Studio.
   b. 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.
2. 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.
   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.

   Consider the following 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 emulate this kernel, you must add a matching `write_channel_altera` 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
```

   You must also modify the host application to create and start this conditional kernel during emulation.

Alternatively, you can 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
    }
}
```

Note: The Altera Software Development Kit (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.

Emulating Your OpenCL Kernel

To emulate your OpenCL kernel, direct the Altera Offline Compiler (AOC) to compile your kernel and generate an x86-64 executable version of the Altera Offline Compiler Executable file (**.aocx**). Run the emulation **.aocx** file on the platform on which you build your kernel.

To compile your OpenCL kernel for emulation, perform the following steps:

1. To generate a **.aocx** file for emulation that targets a specific accelerator board, invoke the `aoc -march=emulator <your_kernel_filename>.cl` command.

   Tip: You have the option to specify the target acclerator board for emulation by including the `--board <board_name>` option in your `aoc -march=emulator` command.

   Attention: You must 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 -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 which libraries are necessary for building a host application. The AOC lists the libraries for both emulation and nonemulation compilation flows.
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, preferrably the current working directory.
5. To run the application, invoke the command `env CL_CONTEXT_EMULATOR_DEVICE_ALTERA=<board_name> <your_host_program_name>`, where <*board_name*> is the FPGA board name you specified when you compiled your kernel.
6. If you change your host or kernel program and you want to test it, only recompile the modified program and rerun emulation.

Each invocation of the emulated kernel creates a shared library copy called ***<process_ID>*-libkernel.so** in the **tmp** folder or directory, where <*process_ID*> is a unique numerical value assigned to each emulation run.

Emulating Systems with Multiple Devices

In a system with multiple devices, kernels on different boards cannot interact with each other directly. Threrefore, in most cases, you can simply emulate one device at a time to verify kernel functionality. There are times when a kernel on one board processes data that another board generates. In these situations, you might have to run kernels from multiple boards in the same emulation session to verify kernel functionality efficiently.

To emulate kernels from multiple boards in one session, model the boards by creating a command queue for each one on the same board.

Below is a code example on how to emulate a system with multiple devices:

```
for( unsigned i=0; i < num_devices; i++ )
{
    // create a command queue
    #ifdef EMULATOR
        int dev_id = 0;
    #else
        int dev_id = i;
    #endif
    queue[i] = clCreateCommandQueue(ctx,
                                    device[dev_id],
                                    CL_QUEUE_PROFILING_ENABLE,
                                    &s);
    checkError(s,"Failed clCreateCommandQueue : queue[i]");
}
```

Caution: If you create command queues for multiple devices, but you set the macro EMULATOR for a single device, the command queue will map to a single device.

Remember: EMULATOR is not an Altera Software Development Kit (SDK) for OpenCL (AOCL) predefined macro. You must explicitly set the EMULATOR macro definition.

Debugging Your OpenCL Kernel (Linux)

For Linux systems, you can direct the Altera Software Development Kit (SDK) for OpenCL 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 in 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. To debug only your host code, use 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, preferrably the current working directory.
5. To run the application, invoke the command `env CL_CONTEXT_EMULATOR_DEVICE_ALTERA=<board_name> gdb --args <your_host_program_name>`, where <*board_name*> is the FPGA board name you specified when you compiled your kernel.
6. If you change your host or kernel program and you want to test it, only recompile the modified program and rerun the debugger.

Limitations of the AOCL Emulator

The Altera Software Development Kit (SDK) for OpenCL (AOCL) emulator is a beta feature with 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. Language support

   Most current debuggers do not support OpenCL-specific features. To set explicitly the receognized language of the supported GNU Project Debugger (GDB) version to OpenCL (GDB version 7.3 or higher), in the debugger, type the `set lang opencl` command.

   For example, when you set the language to OpenCL, the debugger can recognize vector syntax such as `vec.sl` or `vec.xxyz` at the command prompt.
3. Concurrent execution

   Modelling 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.

4. Kernel performance

   The Altera Offine 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.
5. 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.
6. Emulation of channel behavior has limitations, especially for conditional channel operations where the kernel does not call the channel operation in every loop iteration. Therefore, the emulator might execute channel operations in a different order than on the hardware.

Support Statuses of OpenCL Features

A

2014.06.30

OCL002-14.0.0

Subscribe Send Feedback

The Altera Software Development Kit (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 Software Development Kit (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 a clarification for the supported feature is available 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.

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.
	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.

© 2014 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
6.1.2	*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.
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.

Section	Feature	Support Status	Notes
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 *Kernel Pragmas and Attributes* section for more information about these attribute names. Refer to the *Altera SDK for OpenCL Optimization 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.
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 instantiated 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.

Section	Feature	Support Status	Notes
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: *Miscellaneous 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

- **Pragmas and Attributes** on page 1-41
- **Altera SDK for OpenCL Best Practices Guide**
- **OpenCL Specification version 1.2**

OpenCL Programming Language Restrictions

The Altera Software Development Kit (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.

Argument Types for Built-in Geometric Functions

The Altera Software Development Kit (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 the OpenCL compliant devices must support. The Altera Software Development Kit (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.
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	

Section	Feature	Support Status	Notes
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 Software Development Kit (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 Software Development Kit (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 Software Development Kit (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.

Clause	Feature	Support Status	Notes
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
Maximum number of concurrently running kernels	The total number of queues
Maximum number of enqueued kernels	1000
Maximum number of kernels per FPGA device	32
Maximum number of arguments per kernel	128
Maximum total size of kernel arguments	256 bytes per kernel

Document Revision History

B

2014.06.30

OCL002-14.0.0 Subscribe Send Feedback

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 Acesses* 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

© 2014 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

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

November 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*.
November 2012	12.1.0	Initial release.