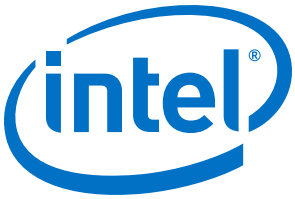


Intel® FPGA SDK for OpenCL™ Pro Edition

Programming Guide

Updated for Intel® Quartus® Prime Design Suite: **18.1.1**

Subscribe
Send Feedback

UG-OCL002 | 2018.12.24
Latest document on the web: PDF | HTML

Contents

1. Intel® FPGA SDK for OpenCL™ Overview

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

1.1. Intel FPGA SDK for OpenCL Pro Edition Programming Guide Prerequisites

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

Before using the Intel FPGA SDK for OpenCL or the Intel FPGA Runtime Environment (RTE) for OpenCL to program your device, familiarize yourself with the respective getting started guides. This document assumes that you have performed the following tasks:

- For developing and deploying OpenCL kernels, download the tar file and run the installers to install the SDK, the Intel Quartus® Prime Pro Edition software, and device support.
- For deployment of OpenCL kernels, download and install the RTE.
- If you want to use the SDK or the RTE to program an Intel SoC FPGA, you also have to download and install the IntelSoC FPGA Embedded Development Suite (EDS) Pro Edition.
- Install and set up your FPGA board.
- Verify that board installation is successful, and the board functions correctly.

If you have not performed the tasks described above, refer to the SDK's getting starting guides for more information.

Related Information

- OpenCL References Pages
- OpenCL Specification version 1.0
- Intel FPGA SDK for OpenCL Pro Edition Getting Started Guide

---

(1) The Intel FPGA 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.

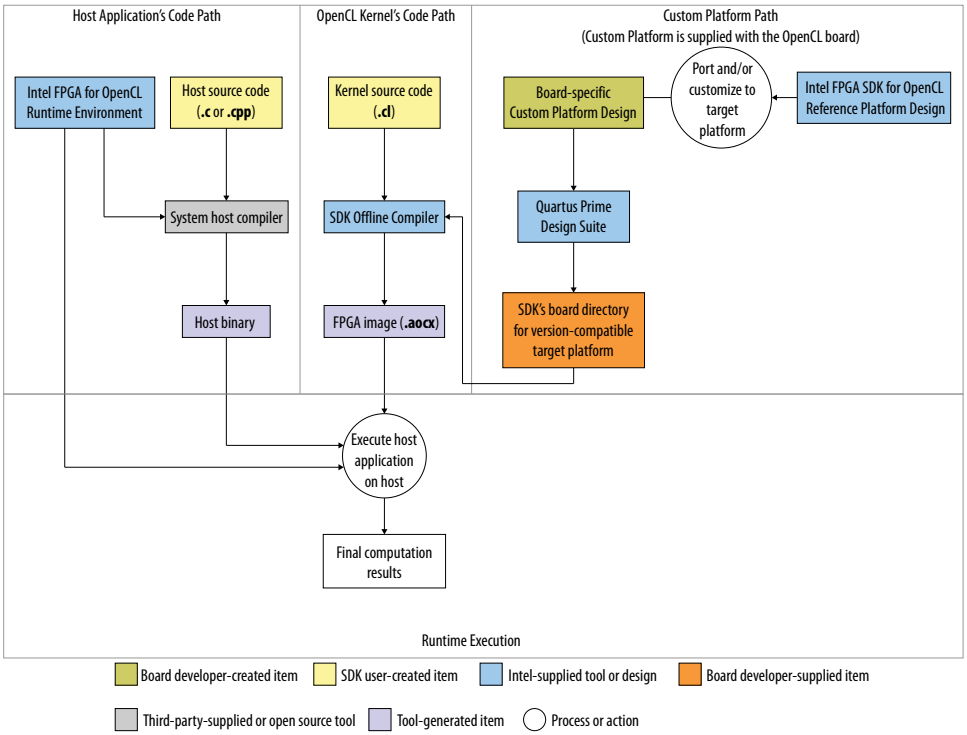
(2) OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission of the Khronos Group™.

- Intel FPGA RTE for OpenCL Pro Edition Getting Started Guide

1.2. Intel FPGA SDK for OpenCL FPGA Programming Flow

Applications using the Intel FPGA SDK for OpenCL have two main components: the FPGA programming bitstream(s), and the host program that manages the application and FPGA accelerator. The Intel FPGA SDK for OpenCL Offline Compiler first compiles your OpenCL kernels to an image file that the host program uses to program the FPGA. The host-side C compiler compiles your host program and then links it to the Intel FPGA SDK for OpenCL runtime libraries.

Figure 1. Schematic Diagram of the Intel FPGA SDK for OpenCL Programming Model



The following SDK components work together to program an Intel FPGA:

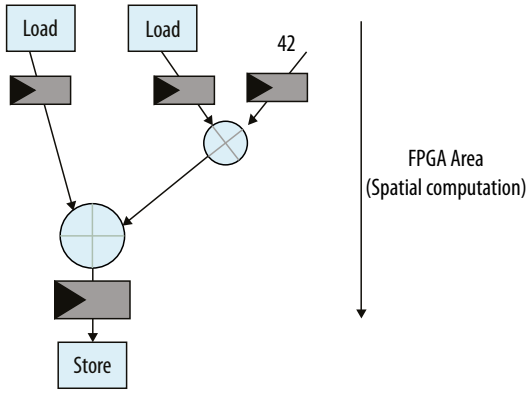
- The host application and the host compiler
- The OpenCL kernel(s) and the offline compiler
- The Custom Platform

The Custom Platform provides the board support package. Typically, the board manufacturer develops the Custom Platform that supports a specific OpenCL board. The offline compiler targets the Custom Platform when compiling an OpenCL kernel to generate a hardware programming image. The host then runs the host application, which usually programs and executes the hardware image onto the FPGA.

In a sequential implementation of a program (for example, on a conventional processor), the program counter controls the sequence of instructions that are executed on the hardware, and the instructions that execute on the hardware across time. In a spatial implementation of a program, such as program implementation

within the Intel FPGA SDK for OpenCL, instructions are executed as soon as the prerequisite data is available. Programs are interpreted as a series of connections representing the data dependencies.

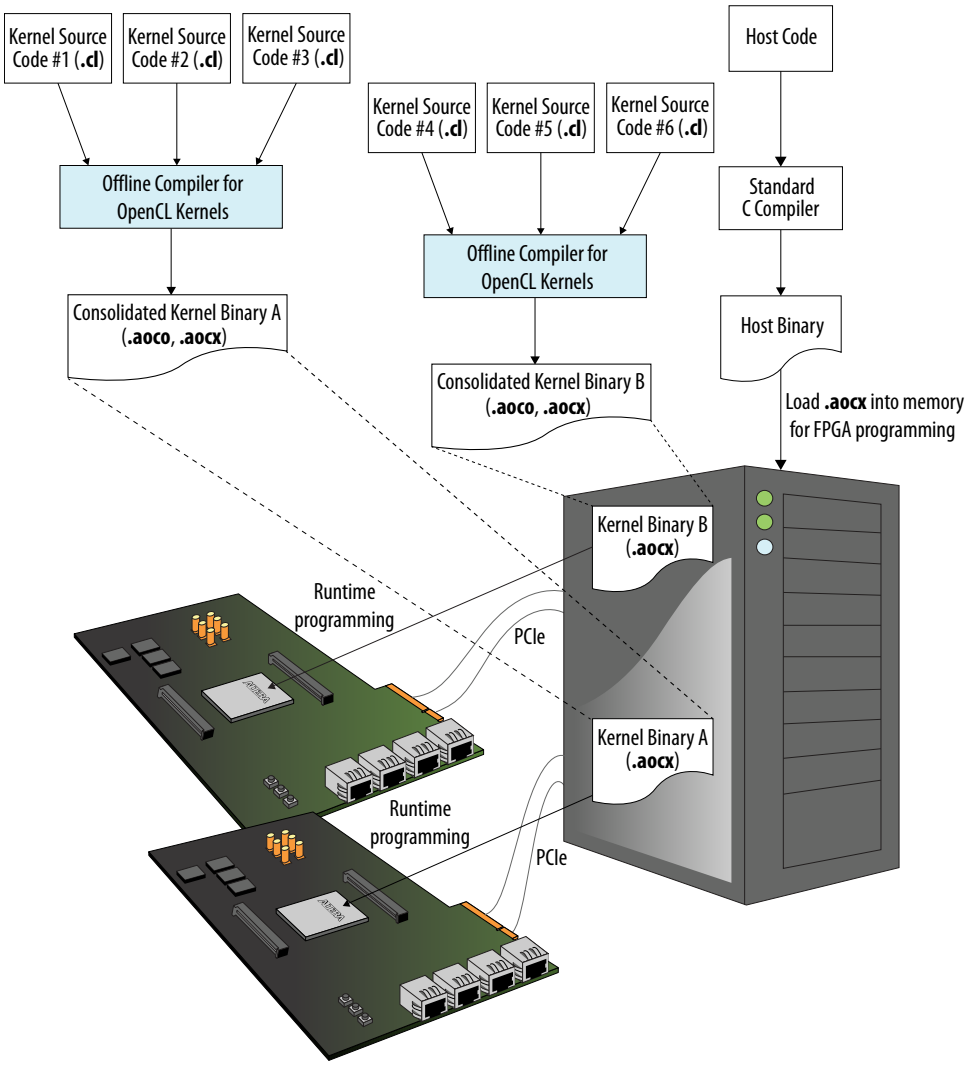
Figure 2. FPGA Data Flow Architecture



2. Intel FPGA SDK for OpenCL Offline Compiler Kernel Compilation Flows

The Intel FPGA SDK for OpenCL Offline Compiler can create your FPGA programming image file (`.aocx` file) in a single or multistep process. The complexity of your kernels dictates the preferred compilation.

Figure 3. The Intel FPGA SDK for OpenCL FPGA Programming Flow



An OpenCL kernel source file (`.cl`) contains your OpenCL kernel source code that runs on the FPGA. The offline compiler groups one or more kernels into a temporary file and then compiles this file to generate the following files and folders:

- A `.aoco` object file is an intermediate object file that contains information for later stages of the compilation.
- A `.aocx` image file is the hardware configuration file and contains information necessary to program the FPGA at runtime.
- The work folder or subdirectory, which contains data necessary to create the `.aocx` file. By default, the name of the work directory is the name of your `.cl` file. If you compile multiple kernel source files, the name of the work directory is the name of the last `.cl` file you list in the `aoc` command line.

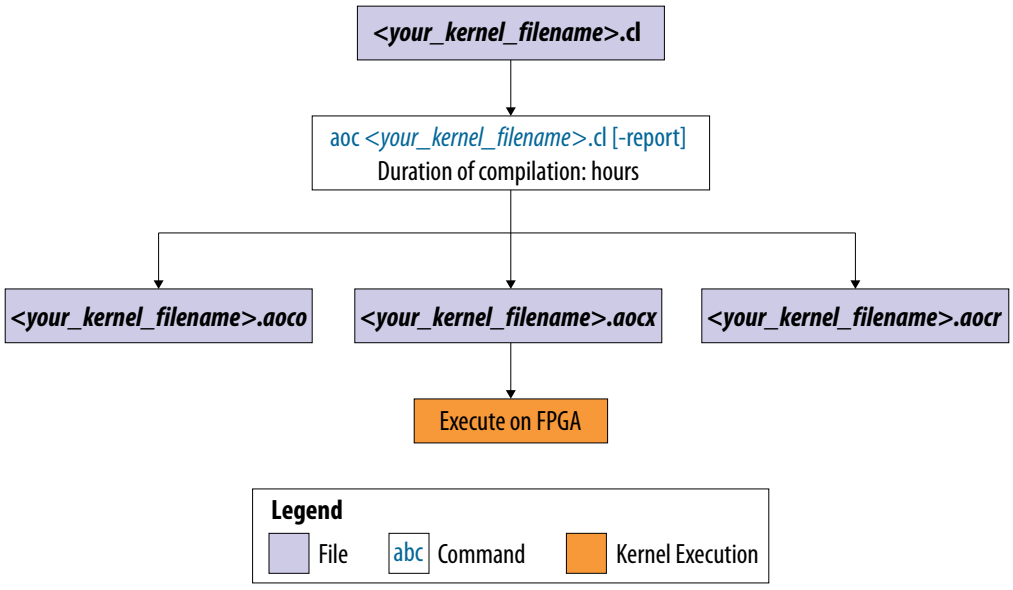
The `.aocx` file contains data that the host application uses to create program objects, a concept within the OpenCL runtime API, for the target FPGA. The host application first loads these program objects into memory. Then the host runtime uses these program objects to program the target FPGA, as required for kernel launch operations by the host program.

2.1. One-Step Compilation for Simple Kernels

By default, the Intel FPGA SDK for OpenCL Offline Compiler compiles one or more OpenCL kernels and creates a hardware configuration file in a single step. Choose this compilation option if you have a single OpenCL kernel in your application, or if you have multiple kernels in your application that require minimal iterations.

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

Figure 4. One-Step OpenCL Kernel Compilation Flow



A successful compilation results in the following files and reports:

- A `.aoco` file
- A `.aocr` file
- A `.aocx` file
- In the `<your_kernel_filename>/reports/report.html` file, the estimated resource usage summary provides a preliminary assessment of area usage. If you have a single work-item kernel, the optimization report identifies performance bottlenecks.

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

Related Information

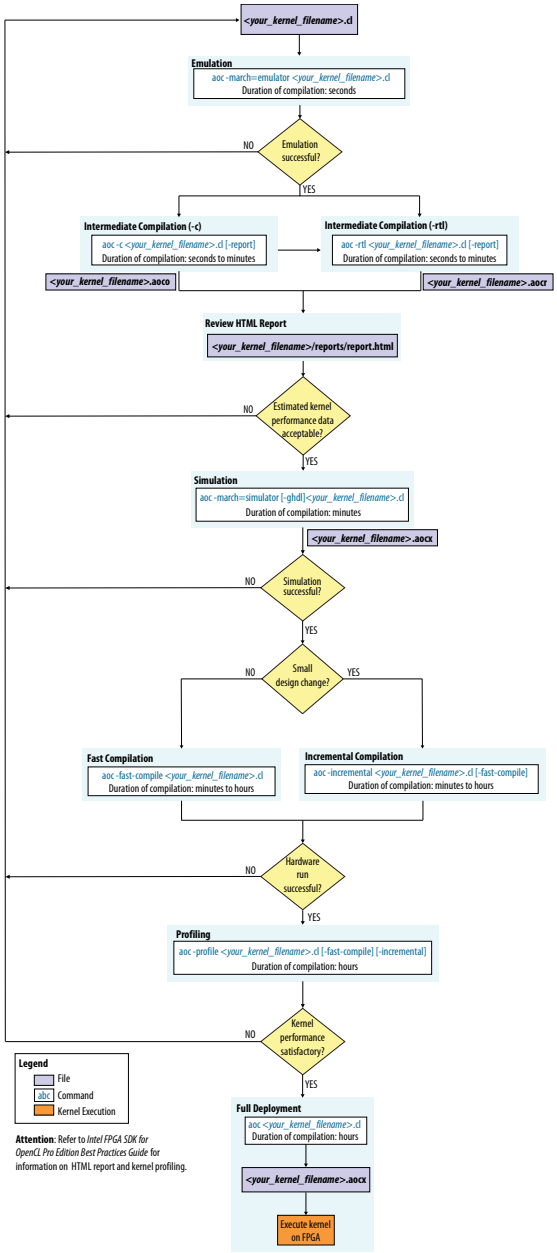
Compiling Your Kernel to Create Hardware Configuration File on page 104

2.2. Multistep Intel FPGA SDK for OpenCL Pro Edition Design Flow

Choose the multistep Intel FPGA SDK for OpenCL Pro Edition design flow if you want to iterate on your OpenCL kernel design to implement optimizations or other iterative modifications.

The following figure outlines the stages in the SDK's design flow. The steps in the design flow serve as checkpoints for identifying functional errors and performance bottlenecks. They allow you to modify your OpenCL kernel code without performing a full compilation on each iteration. You have the option to perform some or all of the compilations steps.

Figure 5. Multistep Intel FPGA SDK for OpenCL Pro Edition Design Flow



The SDK's design flow includes the following steps:

1. Emulation

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

2. Intermediate Compilation

There are two available intermediate compilation steps. You have the option to include one or both of these compilation steps in your design flow.

- Compile one or more `.cl` kernel source files using the `-c` flag. Doing so instructs the offline compiler to generate `.aoco` object files that contain the output from the OpenCL parser.
- Compile one or more `.cl` kernel source files or `.aoco` files, but not both, using the `-rtl` flag. Doing so instructs the offline compiler to perform the following tasks:
  - If the input files are `.cl` files, the offline compiler generates a `.aoco` file for each kernel source file and then links them to generate a `.aocr` file.
  - If the input files are `.aoco` files, the offline compiler links them to generate a `.aocr` file.
  - Creates a <*your_kernel_filename*> directory.

  The offline compiler uses the `.aocr` file to generate the final `.aocx` hardware configuration file.

  Note: If you compile your kernel(s) using the `-c` flag in an environment where the default board is X, and then you compile your `.aoco` files using the `-rtl` flag in an environment where the default board is Y, the offline compiler will read board X from the `.aoco` files and then pass it on to the subsequent compilation stages.

3. Review HTML Report

   Review the <*your_kernel_filename*>`/reports/report.html` file of your OpenCL application to determine whether the estimated kernel performance data is acceptable. The HTML report also provides suggestions on how you can modify your kernel to increase performance.

4. Simulation (Preview)

   Asses the functionality of your OpenCL kernel by running it through simulation. Simulation lets you asses the function correctness and dynamic performance of your kernel without a long compilation time. You can capture and view waveforms for your kernel to help you debug your kernel.

5. Fast Compilation

   Assess the functionality of your OpenCL kernel in hardware. The fast compilation step generates a `.aocx` file in a fraction of the time required to complete a full compilation. The Intel FPGA SDK for OpenCL Offline Compiler reduces compilation time by performing only light optimizations.

6. Incremental Compilation

   Assess the functionality of your OpenCL kernel in hardware. The incremental compilation step generates a `.aocx` file by compiling only the kernels you have modified. The Intel FPGA SDK for OpenCL Offline Compiler improves your productivity by scaling compilation times with the size of your design changes rather than the size of your overall design.

7. Profiling

Instruct the Intel FPGA SDK for OpenCL Offline Compiler to insert performance counters in the FPGA programming image. During execution, the counters collect performance information which you can then review in the Intel FPGA Dynamic Profiler for OpenCL GUI.

8. Full deployment

   When you are satisfied with the performance of your OpenCL kernel throughout the design flow, perform a full compilation. The resulting `.aocx` file will be suitable for deployment.

For more information on the HTML report and kernel profiling, refer to the *Intel FPGA SDK for OpenCL Pro Edition Best Practices Guide.*

Related Information

- Compiling Your Kernel Incrementally (-incremental) on page 115
- Speeding Up Your OpenCL Compilation (-fast-compile) on page 115
- Reviewing Your Kernel's report.html File on page 134
- Compiling Your OpenCL Kernel on page 104
- Emulating and Debugging Your OpenCL Kernel on page 121
- Debugging Your OpenCL Library Through Simulation (Preview) on page 161
- Profiling Your OpenCL Kernel on page 135

3. Obtaining General Information on Software, Compiler, and Custom Platform

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

Notice:

- The Intel FPGA SDK for OpenCL Offline Compiler command options (`aoc <command_option>`) now have single dashes (-) instead of double dashes (--). The double-dash convention was deprecated in the 17.1 release and will be removed in a future release.
- The Intel FPGA SDK for OpenCL Offline Compiler command options now follow the convention `<command_option>=<value>`, where `value` can be a comma separated list of user input values. The use of `-option value1 -option value2` was deprecated in the 17.1 release and will be removed in a future release.

3.1. Displaying the Software Version (version)

To display the version of the Intel FPGA SDK for OpenCL, invoke the `version` utility command.

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

  ```
  aocl <version>.<build> (Intel(R) FPGA SDK for OpenCL(TM),
  Version <version> Build <build>, Copyright (C) <year> Intel
  Corporation)
  ```

3.2. Displaying the Compiler Version (-version)

To display the version of the Intel FPGA SDK for OpenCL Offline Compiler, invoke the `-version` compiler command.

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

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

3.3. Listing the Intel FPGA SDK for OpenCL Utility Command Options (help)

To display information on the Intel FPGA SDK for OpenCL utility command options, invoke the `help` utility command.

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

3.3.1. Displaying Information on an Intel FPGA SDK for OpenCL Utility Command Option (help <command_option>)

To display information on a specific Intel FPGA SDK for OpenCL utility command option, include the command option as an argument of the `help` utility command.

- At a command prompt, invoke the `aocl help` *<command_option>* command.

  For example, to obtain more information on the `install` utility command option, invoke the `aocl help install` command.

  Example output:

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

Usage: aocl install

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

3.4. Listing the Intel FPGA SDK for OpenCL Offline Compiler Command Options (no argument, -help, or -h)

To display information on the Intel FPGA SDK for OpenCL Offline Compiler command options, invoke the compiler command without an argument, or invoke the compiler command with the `-help` or `-h` command option.

- At a command prompt, invoke one of the following commands:
  - `aoc`
  - `aoc -help`
  - `aoc -h`

  The SDK categorizes the offline compiler command options based on their functions. It also provides a description for each option.

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

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

- At a command prompt, invoke the `aoc -list-boards` command.

  The Intel FPGA SDK for OpenCL Offline Compiler generates an output that resembles the following:

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

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

3.6. Displaying the Compilation Environment of an OpenCL Binary (env)

To display the Intel FPGA SDK for OpenCL Offline Compiler's input arguments and the environment for a compiled OpenCL design, invoke the `env` utility command.

- At the command prompt, invoke the `aocl env` *<object_file_name>* or the `aocl env` *<executable_file_name>* command,

  where *<object_file_name>* is the name of the `.aoco` file of your OpenCL kernel, and the *<executable_file_name>* is the name of the `.aocx` file of your kernel.

  Output for the example command `aocl env vector_add.aocx`:

```
INPUT_ARGS=-march=emulator -v device/vector_add.cl -o bin/vector_add.aocx
BUILD_NUMBER=90
ACL_VERSION=16.1.0
OPERATING_SYSTEM=linux
PLATFORM_TYPE=s5_net
```

4. Managing an FPGA Board

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

You can install multiple Custom Platforms simultaneously on the same system with the `aocl install` utility. The Custom Platform subdirectory contains the `board_env.xml` file.

In a system with multiple Custom Platforms, ensure that the host program uses the FPGA Client Driver (FCD), formerly Altera Client Driver (ACD), to discover the boards rather than linking to the Custom Platform memory-mapped device (MMD) libraries directly.

FCD is set up for you when you run the `aocl install` utility. The installed BSP is registered on the system so the runtime and SDK utilities can find the necessary BSP files.

Important: Do not move a BSP to a different directory after you install it. To move a BSP:

1. Uninstall the BSP from its current location with the `aocl uninstall` utility.
2. Change the BSP directory.
3. Reinstall the BSP in the new location with the `aocl install` utility.

Installing an FPGA Board (install) on page 18

Uninstalling an FPGA Board (uninstall) on page 20

Querying the Device Name of Your FPGA Board (diagnose) on page 21

Running a Board Diagnostic Test (diagnose <device_name>) on page 21

Programming the FPGA Offline or without a Host (program <device_name>) on page 22

Programming the Flash Memory (flash <device_name>) on page 22

Related Information

- Installing an FPGA Board (install) on page 18
- Linking Your Host Application to the Khronos ICD Loader Library on page 94

4.1. Installing an FPGA Board (install)

Before creating an OpenCL application for an FPGA accelerator board or SoC device, you must first download and install the Custom Platform from your board vendor. Most Custom Platform installers require administrator privileges. To install your board into the host system, invoke the `aocl install` <*path_to_customplatform*> utility command.

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

Attention: If you are installing the Intel Arria® 10 SoC Development Kit for use with the Intel Arria 10 SoC Development Kit Reference Platform (a10soc), refer to the *Installing the Intel Arria 10 Development Kit* for more information.

Attention: If you want to use Intel FPGA SDK for OpenCL with the Intel Arria 10 GX FPGA Development Kit, refer to the Application Note *AN 807: Configuring the Intel Arria 10 GX FPGA Development Kit for the Intel FPGA SDK for OpenCL* for more information.

1. Follow your board vendor's instructions to connect the FPGA board to your system.
2. Download the Custom Platform for your FPGA board from your board vendor's website. To download an Intel FPGA SDK for OpenCL Reference Platform, refer to the Intel FPGA SDK for OpenCL FPGA Platforms page.

   Tip: If you are installing a BSP that is provided with the Intel FPGA SDK for OpenCL (such as a10_ref, s10_ref, or a10soc, you do not need to download and install a Custom Platform. The BSP files are in *INTELFPGAOCLSDKROOT*/hld/boards. You can skip to 5 on page 19.
3. Install the Custom Platform in a folder that you own (that is, not a system folder).

   You can install multiple Custom Platforms simultaneously on the same system using the SDK utilities, such as `aocl diagnose` with multiple Custom Platforms. The Custom Platform subdirectory contains the `board_env.xml` file.

   In a system with multiple Custom Platforms, ensure that the host program uses the FPGA Client Driver (FCD) to discover the boards rather than linking to the Custom Platforms' memory-mapped device (MMD) libraries directly. As long as FCD is correctly set up for Custom Platform, FCD finds all the installed boards at runtime.
4. Install the Custom Platform in a directory that you own (that is, not a system directory).
5. Set the *QUARTUS_ROOTDIR_OVERRIDE* user environment variable to point to the Intel Quartus Prime Pro Edition software installation directory.
6. Add the paths to the Custom Platform libraries (for example, the memory-mapped (MMD) library) to the *PATH* (Windows) or *LD_LIBRARY_PATH* (Linux) environment variable setting.

   The *Intel FPGA SDK for OpenCL Pro Edition Getting Started Guide* contains more information on the `init_opencl` script. For information on setting user environment variables and running the `init_opencl` script, refer to the *Setting the Intel FPGA SDK for OpenCL Pro Edition User Environment Variables* section.
7. Invoke the command `aocl install` <*path_to_customplatform*> at a command prompt.

   Invoking `aocl install` <*path_to_customplatform*> also installs a board driver that allows communication between host applications and hardware kernel programs.

Remember: You need administrative rights to install a board. To run a Windows command prompt as an administrator, click **Start ➤ All Programs ➤ Accessories**. Under **Accessories**, right click **Command Prompt**, In the right-click menu, click **Run as Administrator**.

On Windows 8.1 or Windows 10 systems, you might also need to disable signed driver verification. For details, see the following articles:

- Windows 8: https://www.intel.com/content/altera-www/global/en_us/index/support/support-resources/knowledge-base/solutions/fb321729.html
- Windows 10: https://www.intel.com/content/altera-www/global/en_us/index/support/support-resources/knowledge-base/embedded/2017/Why-does-aocl-diagnose-fail-while-using-Windows-10.html

8. To query a list of FPGA devices installed in your machine, invoke the `aocl diagnose` command.
   The software generates an output that includes the <*device_name*>, which is an acl number that ranges from acl0 to acl127.

   Attention: For possible errors after implementing the `aocl diagnose` utility, refer to Possible Errors After Running the `diagnose` Utility section in the *Intel Arria 10 GX FPGA Development Kit Reference Platform Porting Guide*. For more information on querying the <*device_name*> of your accelerator board, refer to the *Querying the Device Name of Your FPGA Board* section.

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

Related Information

- Querying the Device Name of Your FPGA Board (diagnose) on page 21
- Setting the Intel FPGA SDK for OpenCL Pro Edition User Environment Variables (Windows)
- Setting the Intel FPGA SDK for OpenCL Pro Edition User Environment Variables (Linux)
- Intel FPGA SDK for OpenCL FPGA Platforms page

4.2. Uninstalling an FPGA Board (uninstall)

To uninstall an FPGA board, invoke the `uninstall` utility command, uninstall the Custom Platform, and unset the relevant environment variables.

To uninstall your FPGA board, perform the following tasks:

1. Disconnect the board from your machine by following the instructions provided by your board vendor.
2. Invoke the `aocl uninstall` *<path_to_customplatform>* utility command to remove the current host computer drivers (for example, PCIe® drivers). The Intel FPGA SDK for OpenCL uses these drivers to communicate with the FPGA board.
3. Uninstall the Custom Platform.
4. Unset the *LD_LIBRARY_PATH* (for Linux) or *PATH* (for Windows) environment variable.

4.3. Querying the Device Name of Your FPGA Board (diagnose)

Some OpenCL software utility commands require you to specify the device name (*<device_name>*). The *<device_name>* refers to the acl number (e.g. acl0 to acl127) that corresponds to the FPGA device. When you query a list of accelerator boards, the OpenCL 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, type `aocl diagnose` at a command prompt.
  The software generates an output that resembles the example shown below:

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

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

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

device_name  Status  Information

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

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

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

DIAGNOSTIC_PASSED
```

Related Information

Probing the OpenCL FPGA Devices on page 98

4.4. Running a Board Diagnostic Test (diagnose <device_name>)

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

- At a command prompt, invoke the `aocl diagnose` *<device_name>* command, where *<device_name>* is the acl number (for example, acl0 to acl127) that corresponds to your FPGA device.

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

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

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

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

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

  where:

  <device_name> refers to the acl number (for example, acl0 to acl127) that corresponds to your FPGA device, and

  <your_kernel_filename>`.aocx` is the executable file you use to program the hardware.

Note: To program an SoC, specify the full path of the device when invoking the `program` utility command. For example, `aocl program /dev/`*<device_name>* *<your_kernel_filename>*`.aocx`.

4.6. Programming the Flash Memory (flash <device_name>)

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

Note: For example instructions on programming the micro SD flash card on an SoC board such as the Intel Arria 10 SoC Development Kit, refer to the *Building the SD Card Image* section of the *Intel FPGA SDK for OpenCL Intel Arria 10 SoC Development Kit Reference Platform Porting Guide*.

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

  where:

  <device_name> refers to the acl number (for example, acl0 to acl127) that corresponds to your FPGA device, and

  <your_kernel_filename>`.aocx` is the executable file you use to program the hardware.

Related Information

Building the SD Card Image for the Intel Arria 10 SoC Development Kit

5. Structuring Your OpenCL Kernel

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

5.1. Guidelines for Naming the Kernel

Intel recommends that you include only alphanumeric characters in your file names.

- Begin a file name with an alphanumeric character.

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

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

- Do not differentiate file names using nonalphanumeric characters.

  The Intel FPGA SDK for OpenCL Offline Compiler translates any nonalphanumeric character into an underscore ("_"). If you differentiate two file names by ending them with different nonalphanumeric characters only (for example, `myKernel#.cl` and `myKernel&.cl`), the offline compiler translates both file names to `<your_kernel_filename>_.cl` (for example, `myKernel_.cl`).

- For Windows systems, ensure that the combined length of the kernel file name and its file path does not exceed 260 characters.

  64-bit Windows 7 and Windows 8.1 have a 260-character limit on the length of a file path. If the combined length of the kernel file name and its file path exceeds 260 characters, the offline compiler generates the following error message:

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

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

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

  For Windows 10, you can remove the 260-character limit. For more information, see your Windows 10 documentation.

- Do not name your `.cl` OpenCL kernel source file "kernel", "Verilog", or "VHDL" as they are reserved keywords.

  Naming the source file `kernel.cl`, `Verilog.cl`, or `VHDL.cl` causes the offline compiler to generate intermediate design files that have the same names as certain internal files, which leads to a compilation error.

5.2. Programming Strategies for Optimizing Data Processing Efficiency

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

Specifying Number of SIMD Work-Items on page 31

5.2.1. Unrolling a Loop

Loop unrolling involves replicating a loop body multiple times, and reducing the trip count of a loop. Unroll loops to reduce or eliminate loop control overhead on the FPGA. In cases where there are no loop-carried dependencies and the offline compiler can perform loop iterations in parallel, unrolling loops can also reduce latency and overhead on the FPGA.
The Intel FPGA SDK for OpenCL Offline Compiler might unroll simple loops even if they are not annotated by a pragma.
To direct the offline compiler to unroll a loop, or explicitly not to unroll a loop, insert an `unroll` kernel pragma in the kernel code preceding a loop you want to unroll.

Attention:

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

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

  Consider the code fragment below. By assigning a value of 2 as the unroll factor, you direct the offline compiler to unroll the loop twice.

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

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

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

- To prevent a loop from unrolling, specify an unroll factor of 1 (that is, `#pragma unroll 1`).

5.2.2. Coalescing Nested Loops

Use the `loop_coalesce` pragma to direct the Intel FPGA SDK for OpenCL Offline Compiler to coalesce nested loops into a single loop without affecting the loop functionality. Coalescing loops can help reduce your kernel area usage by directing the compiler to reduce the overhead needed for loop control.

Coalescing nested loops also reduces the latency of the component, which could further reduce your kernel area usage. However, in some cases, coalescing loops might lengthen the critical loop initiation interval path, so coalescing loops might not be suitable for all kernels.

For NDRange kernels, the compiler automatically attempts to coalesce loops even if they are not annotated by the `loop_coalesce` pragma. Coalescing loops in NDRange kernels improves throughput as well as reducing kernel area usage. You can use the `loop_coalesce` pragma to prevent the automatic coalescing of loops in NDRange kernels.

To coalesce nested loops, specify the pragma as follows:

```
#pragma loop_coalesce <loop_nesting_level>
```

The <*loop_nesting_level*> parameter is optional and is an integer that specifies how many nested loop levels that you want the compiler to attempt to coalesce. If you do not specify the <*loop_nesting_level*> parameter, the compiler attempts to coalesce all of the nested loops.

For example, consider the following set of nested loops:

```
for (A)
  for (B)
    for (C)
      for (D)
    for (E)
```

If you place the pragma before loop (A), then the loop nesting level for these loops is defined as:

- Loop (A) has a loop nesting level of 1.
- Loop (B) has a loop nesting level of 2.
- Loop (C) has a loop nesting level of 3.
- Loop (D) has a loop nesting level of 4.
- Loop (E) has a loop nesting level of 3.

Depending on the loop nesting level that you specify, the compiler attempts to coalesce loops differently:

- If you specify `#pragma loop_coalesce 1` on loop (A), the compiler does not attempt to coalesce any of the nested loops.
- If you specify `#pragma loop_coalesce 2` on loop (A), the compiler attempts to coalesce loops (A) and (B).
- If you specify `#pragma loop_coalesce 3` on loop (A), the compiler attempts to coalesce loops (A), (B), (C), and (E).
- If you specify `#pragma loop_coalesce 4` on loop (A), the compiler attempts to coalesce all of the loops [loop (A) - loop (E)].

Important: If you specify `#pragma loop_coalesce 1` for a loop in an NDRange kernel, you prevent automatic loop coalescing for that loop.

Example

The following simple example shows how the compiler coalesces two loops into a single loop.

Consider a simple nested loop written as follows:

```
#pragma loop_coalesce
for (int i = 0; i < N; i++)
  for (int j = 0; j < M; j++)
    sum[i][j] += i+j;
```

The compiler coalesces the two loops together so that they run as if they were a single loop written as follows:

```
int i = 0;
int j = 0;
while(i < N){

  sum[i][j] += i+j;
  j++;

  if (j == M){
    j = 0;
    i++;
  }
}
```

5.2.3. Specifying a Loop Initiation interval (II)

The initiation interval, or II, is the number of clock cycles between the launch of successive loop iterations. Use the `ii` pragma to direct the Intel FPGA SDK for OpenCL Offline Compiler to attempt to set the initiation interval (II) for the loop that follows the pragma declaration. If the offline compiler cannot achieve the specified II for the loop, then the compilation errors out.

The `ii` pragma applies to single work-item kernels (that is, single-threaded kernels) in which loops are pipelined. Refer to the *Single Work-Item Kernel versus NDRange Kernel* section of the *Intel FPGA SDK for OpenCL Best Practices Guide* for information on loop pipelining, and on kernel properties that drive the offline compiler's decision on whether to treat a kernel as single-threaded.

The higher the II value, the longer the wait before the subsequent loop iteration starts executing. Refer to the *Reviewing Your Kernel's report.html File* section of the *Intel FPGA SDK for OpenCL Best Practices Guide* for information on II, and on the compiler reports that provide you with details on the performance implications of II on a specific loop.

For some loops in your kernel, specifying a higher II value with the `ii` pragma than the value the compiler chooses by default can increase the maximum operating frequency (f_{max}) of your kernel without a decrease in throughput.

A loop is a good candidate to have the `ii` pragma applied to it if the loop meets the following conditions:

- The loop is pipelined because the kernel is single-threaded.
- The loop is not critical to the throughput of your kernel.
- The running time of the loop is small compared to other loops it might contain.

To specify a loop initiation interval for a loop, specify the pragma before the loop as follows:

```
#pragma ii <desired_initiation_interval>
```

The *<desired_initiation_interval>* parameter is required and is an integer that specifies the number of clock cycles to wait between the beginning of execution of successive loop iterations.

Example

Consider a case where your kernel has two distinct, pipelineable loops: a short-running initialization loop that has a loop-carried dependence and a long-running loop that does the bulk of your processing. In this case, the compiler does not know that the initialization loop has a much smaller impact on the overall throughput of your design. If possible, the compiler attempts to pipeline both loops with an II of 1.

Because the initialization loop has a loop-carried dependence, it will have a feedback path in the generated hardware. To achieve an II with such a feedback path, some clock frequency might be sacrificed. Depending on the feedback path in the main loop, the rest of your design could have run at a higher operating frequency.

If you specify `#pragma ii 2` on the initialization loop, you tell the compiler that it can be less aggressive in optimizing II for this loop. Less aggressive optimization allows the compiler to pipeline the path limiting the f_{max} and could allow your overall kernel design to achieve a higher f_{max}.

The initialization loop takes longer to run with its new II. However, the decrease in the running time of the long-running loop due to higher f_{max} compensates for the increased length in running time of the initialization loop.

Related Information

- Single Work-Item Kernel versus NDRange Kernel
- Reviewing Your Kernel's report.html File

5.2.4. Loop Concurrency (`max_concurrency` Pragma)

You can use the `max_concurrency` pragma to decrease the concurrency of a loop in your component. The concurrency of a loop is how many iterations of that loop can be in progress at one time. By default, the Intel FPGA SDK for OpenCL tries to maximize the concurrency of loops so that your component runs at peak throughput.

The `max_concurrency` pragma applies to single work-item kernels (that is, single-threaded kernels) in which loops are pipelined. Refer to the *Single Work-Item Kernel versus NDRange Kernel* section of the *Intel FPGA SDK for OpenCL Pro Edition Best Practices Guide* for information on loop pipelining, and on kernel properties that drive the offline compiler's decision on whether to treat a kernel as single-threaded.

The `max_concurrency` pragma enables you to control the on-chip memory resources required to implement your loop. To achieve simultaneous execution of loop iterations, the offline compiler must create independent copies of any memory that is private to a single iteration. The greater the permitted concurrency, the more copies the compiler must make.

The kernel's HTML report (`report.html`) provides the following information pertaining to loop concurrency:

- Maximum concurrency that the offline compiler has chosen

  This information is available in the Loop Analysis report. A message in the Details pane reports that the maximum number of simultaneous executions has been limited to *N*.

- Impact to memory usage

  This information is available in the Area Analysis report. A message in the Details pane reports that the offline compiler has created *N* independent copies of the memory to enable simultaneous execution of *N* loop iterations.

If you want to exchange some performance for physical memory savings, apply `#pragma max_concurrency <N>` to the loop, as shown below. When you apply this pragma, the offline compiler limits the number of simultaneously-executed loop iterations to *N*. The number of independent copies of loop memories is also reduced to *N*.

```
#pragma max_concurrency 1
for (int i = 0; i < N; i++) {
  int arr[M];
  // Doing work on arr
}
```

Related Information

- Single Work-Item Kernel versus NDRange Kernel
- Reviewing Your Kernel's report.html File

5.2.5. Specifying Work-Group Sizes

Specify a maximum or required work-group size whenever possible. The Intel FPGA SDK for OpenCL Offline Compiler relies on this specification to optimize hardware usage of the OpenCL kernel without involving excess logic.

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

- If your kernel contains a barrier, the offline compiler sets a default maximum scalarized work-group size of 256 work-items.
- If your kernel does not query any OpenCL intrinsics that allow different threads to behave differently (that is, local or global thread IDs, or work-group ID), the offline compiler infers a single-threaded execution mode and sets the maximum work-group size to (1,1,1). In this case, the OpenCL runtime also enforces a global enqueue size of (1,1,1), and loop pipelining optimizations are enabled within the offline compiler.

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

- To specify the maximum number of work-items that the offline compiler will provision for a work-group in a kernel, insert the `max_work_group_size(`*X*`, `*Y*`, `*Z*`)` attribute in your kernel source code.

  For example:

```
__attribute__((max_work_group_size(512,1,1)))
__kernel void sum (__global const float * restrict a,
                   __global const float * restrict b,
                   __global float * restrict answer)
{
    size_t gid = get_global_id(0);
    answer[gid] = a[gid] + b[gid];
}
```

- To specify the required number of work-items that the offline compiler provisions for a work-group in a kernel, insert the `reqd_work_group_size(`*X*`, `*Y*`, `*Z*`)` attribute in your kernel source code.

  For example:

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

5.2.6. Specifying Number of Compute Units

To increase the data-processing efficiency of an OpenCL kernel, you can instruct the Intel FPGA SDK for OpenCL Offline Compiler to generate multiple kernel compute units. Each compute unit is capable of executing multiple work-groups simultaneously.

Caution: Multiplying the number of kernel compute units increases data throughput at the expense of FPGA resource consumption and global memory bandwidth contention between compute units.

- To specify the number of compute units for a kernel, insert the `num_compute_units(`*N*`)` attribute in the kernel source code.

  For example, the code fragment below directs the offline compiler to instantiate two compute units in a kernel:

```
__attribute__((num_compute_units(2)))
__kernel void test(__global const float * restrict a,
                   __global const float * restrict b,
                   __global float * restrict answer)
{
```

```
    size_t gid = get_global_id(0);
    answer[gid] = a[gid] + b[gid];
}
```

The offline compiler dynamically distributes work-groups across the specified number of compute units.

Note: To identify the specific compute unit on which a work-item is executing, call the `get_compute_id()` intrinsic function. Refer to *Customization of Replicated Kernels Using the get_compute_id() Function* for more information.

Related Information

Customization of Replicated Kernels Using the get_compute_id() Function on page 173

5.2.7. Specifying Number of SIMD Work-Items

You have the option to increase the data-processing efficiency of an OpenCL kernel by executing multiple work-items in a single instruction multiple data (SIMD) manner without manually vectorizing your kernel code. Specify the number of work-items within a work-group that the Intel FPGA SDK for OpenCL Offline Compiler should execute in an SIMD or vectorized manner.

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

- To specify the number of SIMD work-items in a work-group, insert the `num_simd_work_item(N)` attribute in the kernel source code.

  For example, the code fragment below assigns a fixed work-group size of 64 work-items to a kernel. It then consolidates the work-items within each work-group into four SIMD vector lanes:

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

  The offline compiler vectorizes the kernel datapath according to the value you specify for `num_simd_work_items` whenever possible.

5.3. Programming Strategies for Optimizing Pointer-to-Local Memory Size

When using a pointer-to-local kernel argument to create a local memory allocation, the Intel FPGA SDK for OpenCL Offline Compiler must decide at compilation time (not runtime) on the size of the local memory system to build on the FPGA. You can optimize the memory size by specifying the size that you will request at runtime using

`clSetKernelArg`. This specification allows the offline compiler to build the correctly sized local memory system for the pointer argument. If you do not specify a size, the offline compiler will use the default size.

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

  The value *N* specifies the desired memory size in bytes. For efficiency, *N* should be a power of two.

  For example:

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

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

Attention: Instead of using point-to-local kernel arguments, Intel recommends that you define local memory systems within the kernel scope.

5.4. Implementing the Intel FPGA SDK for OpenCL Channels Extension

The Intel FPGA SDK for OpenCL channels extension provides a mechanism for passing data between kernels and synchronizing kernels with high efficiency and low latency.

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

Related Information

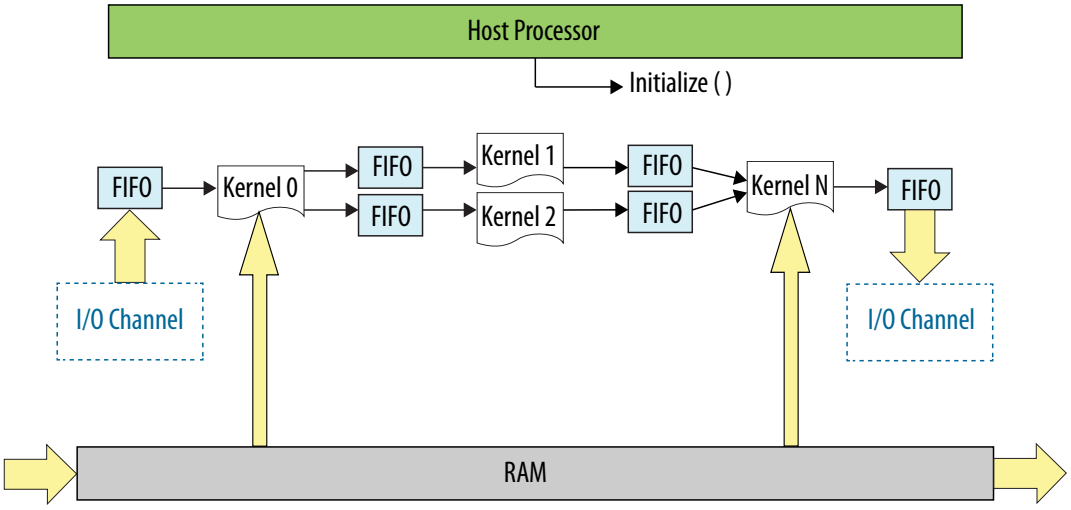
Implementing OpenCL Pipes on page 49

5.4.1. Overview of the Intel FPGA SDK for OpenCL Channels Extension

The Intel FPGA SDK for OpenCL channels extension allows kernels to communicate directly with each other through FIFO buffers.

Implementation of channels decouples data movement between concurrently executing kernels from the host processor.

Figure 6. Overview of Channels Implementation



5.4.2. Channel Data Behavior

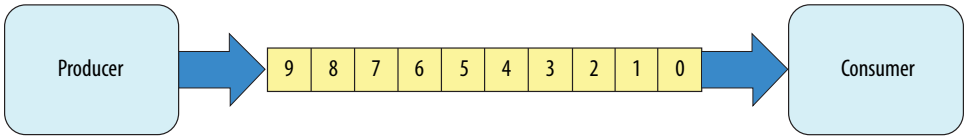
Data written to a channel remains in a channel as long as the kernel program remains loaded on the FPGA device. In other words, data written to a channel persists across multiple work-groups and NDRange invocations. However, data is not persistent across multiple or different invocations of kernel programs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

Data in channels does not persist between context, program, device, kernel, or platform releases, even if the OpenCL implementation performs optimizations that avoid reprogramming operations on a device. For example, if you run a host program twice using the same `.aocx` file, or if a host program releases and reacquires a context, the data in the channel might or might not persist across the operation. FPGA device reset operations might happen behind the scenes on object releases that purge data in any channels

Consider the following code example:

```
channel int c0;

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

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

Figure 7. Channel Data FIFO Ordering



The kernel `producer` writes ten elements ([0, 9]) to the channel. The kernel consumer does not contain any work-item identifier queries; therefore, it will receive an implicit `reqd_work_group_size` attribute of (1,1,1). The implied `reqd_work_group_size(1,1,1)` attribute means that consumer must be launched as a single work-item kernel. In the example above, `consumer` reads five elements from the channel per invocation. During the first invocation, the kernel `consumer` reads values 0 to 4 from the channel. Because the data persists across NDRange invocations, the second time you execute the kernel `consumer`, it reads values 5 to 9.

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

5.4.3. Multiple Work-Item Ordering for Channels

The OpenCL specification does not define a work-item ordering. The Intel FPGA SDK for OpenCL enforces a work-item order to make it easier to rationalize channel read and write operations.

Multiple work-item accesses to a channel can be useful in some scenarios. For example, they are useful when data words in the channel are independent, or when the channel is implemented for control logic. The main concern regarding multiple work-item accesses to a channel is the order in which the kernel writes data to and reads data from the channel. If possible, the SDK's channels extension processes work-item 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 SDK checks that a channel access is work-item invariant based on the following characteristics:

- All paths through the kernel must execute the channel access.
- If the first requirement is not satisfied, none of the branch conditions that reach the channel call should execute in a work-item-dependent manner.
- The kernel is not inferred as a single work-item kernel.

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

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

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

5.4.3.1. Work-Item Serial Execution of Channels

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

When you implement channels in a kernel, the Intel FPGA SDK for OpenCL Offline Compiler enforces that kernel behavior is equivalent to having at most one work-group in flight within the compute unit at a time. The compiler also ensures that the kernel executes channels in work-item serial execution, where the kernel executes work-items with smaller IDs first. A work-item has the identifier `(x, y, z, group)`, where `x`, `y`, `z` are the local 3D identifiers, 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`

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

Channel Execution in Loop with Multiple Work-Items

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

```
__kernel void ordering (__global int * data, int X) {
  int n = 0;
  while (n < X)
  {
    write_channel_intel (req, data[get_global_id(0)]);
    n++;
  }
}
```

5.4.4. Restrictions in the Implementation of Intel FPGA SDK for OpenCL Channels Extension

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

Multiple Channel Call Site

A kernel can read from the same channel multiple times. However, multiple kernels cannot read from the same channel. Similarly, a kernel can write to the same channel multiple times but multiple kernels cannot write to the same channel.

```
__kernel void k1() {
     read_channel_intel (channel1);
     read_channel_intel (channel1);
     read_channel_intel (channel1);
}
```

The Intel FPGA SDK for OpenCL Offline Compiler cannot compile the following code and will issue an error:

```
__kernel void k1(){
     write_channel_intel (channel1, 1);
}

__kernel void k2() {
     write_channel_intel (channel1, 2);
}
```

Feedback and Feed-Forward Channels

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

Static Indexing

The Intel FPGA SDK for OpenCL channels extension does support indexing into arrays of channel IDs, but it leads to inefficient hardware.

Consider the following example:

```
channel int ch[WORKGROUP_SIZE];

__kernel void consumer()
{

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

    //statements
}
```

Compilation of this example generates the following warning message:

```
Compiler Warning: Dynamic access into channel array ch was expanded into
predicated
static accesses on every channel of the array.
```

If the access is dynamic and you know that only a subset of the channels in the array can be accessed, you can generate slightly more efficient hardware with a switch statement:

```
channel int ch[WORKGROUP_SIZE];

__kernel void consumer() {

    int gid = get_global_id(0);
    int value;
```

```
    switch(gid)
    {
        case 0: value = read_channel_intel(ch[0]); break;

        case 2: value = read_channel_intel(ch[2]); break;
        case 3: value = read_channel_intel(ch[3]); break;
        //statements

        case WORKGROUP_SIZE-1:read_channel_intel(ch[WORKGROUP_SIZE-1]); break;
    }

    //statements
}
```

Kernel Vectorization Support

You cannot vectorize kernels that use channels; that is, do not include the `num_simd_work_items` kernel attribute in your kernel code. Vectorizing a kernel that uses channels creates multiple channel accesses inside the same kernel and requires arbitration, which negates the advantages of vectorization. As a result, the SDK's channel extension does not support kernel vectorization.

Instruction-Level Parallelism on read_channel_intel and write_channel_intel Calls

If no data dependencies exist between `read_channel_intel` and `write_channel_intel` calls, the offline compiler attempts to execute these instructions in parallel. As a result, the offline compiler might execute these `read_channel_intel` and `write_channel_intel` 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_intel(channel1);
in_data2 = read_channel_intel(channel2);
in_data3 = read_channel_intel(channel3);
```

Because there are no data dependencies between the `read_channel_intel` calls, the offline compiler can execute them in any order.

5.4.5. Enabling the Intel FPGA SDK for OpenCL Channels for OpenCL Kernel

To implement the Intel FPGA SDK for OpenCL channels extension, modify your OpenCL kernels to include channels-specific pragma and API calls.

To enable the channel extension, use the following pragma:
`#pragma OPENCL EXTENSION cl_intel_channels : enable`

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

5.4.5.1. Declaring the Channel Handle

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

To read from and write to a channel, the kernel must pass the channel variable to each of the corresponding API calls.

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

  For example: `channel int c;`

- The Intel FPGA SDK for OpenCL channel extension supports simultaneous channel accesses by multiple variables declared in a data structure. Declare a `struct` data structure for a channel in the following manner:

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

channel type_t foo;
```

5.4.5.2. Implementing Blocking Channel Writes

The `write_channel_intel` API call allows you to send data across a channel.

- To implement a blocking channel write, use the following `write_channel_intel` function signature:

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

  Where:

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

  `data` is the data that the channel write operation writes to the channel.

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

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

```
//Defines chan, a 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_intel(chan, src[i]);
  }
}
```

Caution: When you send data across a channel using the `write_channel_intel` API call, keep in mind that if the channel is full (that is, if the FIFO buffer is full of data), your kernel will stall and wait until at least one data slot becomes available in the FIFO buffer. Use the Intel FPGA Dynamic Profiler for OpenCL to check for channel stalls.

Related Information

Profiling Your OpenCL Kernel on page 135

5.4.5.2.1. Implementing Nonblocking Channel Writes

Perform nonblocking channel writes to facilitate applications where writes to a full FIFO buffer should not cause the kernel to stall until a slot in the FIFO buffer becomes free. A nonblocking channel write returns a boolean value that indicates whether data was written successfully to the channel (that is, the channel was not full).

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

- To implement a nonblocking channel write, include the following `write_channel_nb_intel` function signature:

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

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

```
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_intel(worker0, src[i]);
      if(!success) {
        success = write_channel_nb_intel(worker1, src[i]);
      }
    }
    while(!success);
  }
}
```

5.4.5.3. Implementing Blocking Channel Reads

The `read_channel_intel` API call allows you to receive data across a channel.

- To implement a blocking channel read, include the following `read_channel_intel` function signature:

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

  Where:

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

  <type> defines a channel data width. Ensure that the variable the kernel assigns to read the channel data is convertible from *<type>*.

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

```
//Defines chan, a 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_intel(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_intel` API call. Doing so causes your kernel to stall until at least one element of data becomes available from the FIFO buffer.

5.4.5.3.1. Implementing Nonblocking Channel Reads

Perform nonblocking reads to facilitate applications where data is not always available and the operation should not wait for data to become available.The nonblocking read signature is similar to a blocking read. However, it populates the address pointed to by the bool pointer `valid` indicating whether a read operation successfully read data from the channel.

On a successful read (`valid` set to true), the value read from the channel is returned by the `read_channel_nb_intel` function. On a failed read (`valid` set to false), the return value of the `read_channel_nb_intel` function is not defined.

- To implement a blocking channel write, use the following `read_channel_nb_intel` function signature:

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

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

```
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_intel(chan, &valid0);
    long data1 = read_channel_nb_intel(chan, &valid1);
    if (valid0) {
       process(data0);
    }
    if (valid1) {
       process(data1);
    }
  }
}
```

5.4.5.4. Implementing I/O Channels Using the io Channels Attribute

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

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

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

Caution:

- Implicit data dependencies might exist for channels that connect to the board directly and communicate with peripheral devices via I/O channels. These implicit data dependencies might lead to unexpected behavior because the Intel FPGA SDK for OpenCL Offline Compiler does not have visibility into these dependencies.
- External I/O channels communicating with the same peripherals do not obey any sequential ordering. Ensure that the external device does not require sequential ordering because unexpected behavior might occur.

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

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

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

   The `width` attribute of an `interface` element specifies the width, in bits, of the data type used by that channel. For the example above, both the `uint` and `float` data types are 32 bits wide. Other bigger or vectorized data types must match the appropriate bit width specified in the `board_spec.xml` file.

2. Implement the `io` channel attribute as demonstrated in the following code example. The `io` channel attribute names must match those of the I/O channels (`chan_id`) specified in the `board_spec.xml` file.

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

   __kernel void io_in_kernel (__global ulong4 *mem_read,
                               uchar read_from,
                               int size)
   {
     int index = 0;
     ulong4 data;
   ```

```
  int half_size = size >> 1;
  while (index < half_size)
  {
    if (read_from & 0x01)
    {
      data = read_channel_intel(udp_in_IO);
    }
    else
    {
      data = mem_read[index];
    }
    write_channel_intel(udp_in, data);
    index++;
  }
}

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

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

5.4.5.5. Emulating I/O Channels

When you emulate a kernel that has a channel declared with the `io` attribute, I/O channel input is emulated by reading from a file, and channel output is emulated by writing to a file.

When you run your emulation, the file name used for reading or writing matches the name in the `io` attribute. For example, if you have a channel declaration as follows, your emulation would read or write (but not both) to a file called `myIOChannel`.

```
channel uint chanA __attribute__((io("myIOChannel")));
```

I/O channels are unidirectional. You can either read from a channel or write to a channel, but not both. However, you can have separate read channels and write channels with the same `io` attribute value.

```
channel uint readChannel __attribute__((io("myIOChannel")));
channel uint writeChannel __attribute__((io("myIOChannel")));
```

Emulating Reading from an I/O Channel

If a read is issued from a channel with an `io` attribute called `myfile`, a read attempt is made on a file on the disk called `myfile`. If the `myfile` file does not exist or there is insufficient data to read from the file, the behavior depends on the call type:

Non-blocking read	If the file does not exist or there is insufficient data, the read attempt returns with a failure message.
Blocking readl	If the file does not exist or there is insufficient data, the read attempt blocks your program until the file is created on the disk, or the file contains sufficient data.

Emulating Writing to an I/O Channel

If a write is issued to a channel with an `io` attribute called `myfile`, a write attempt is made to a file on the disk called `myfile`. If the `myfile` file does not exist, a regular file is created and written to. If the `myfile` file exists, it is overwritten. If the write fails, the behavior depends on the call type:

Non-blocking write	If the write attempt fails, an error is returned.
Blocking write	If the write attempt fails, further write attempts are made.

5.4.5.6. Use Models of Intel FPGA SDK for OpenCL Channels Implementation

Concurrent kernel execution can improve the efficiency of channels on an FPGA. To achieve concurrent execution, the host launches the kernels in parallel. The kernels can communicate with each other through channels where applicable.

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

Feed-Forward Design Model

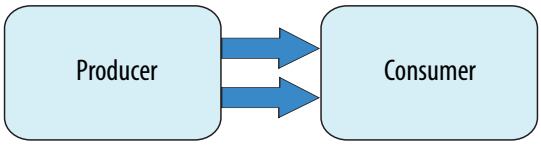
Implement the feed-forward design model to send data from one kernel to the next without creating any cycles between them. Consider the following code example:

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

        write_channel_intel(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_intel(c0);
```

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

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

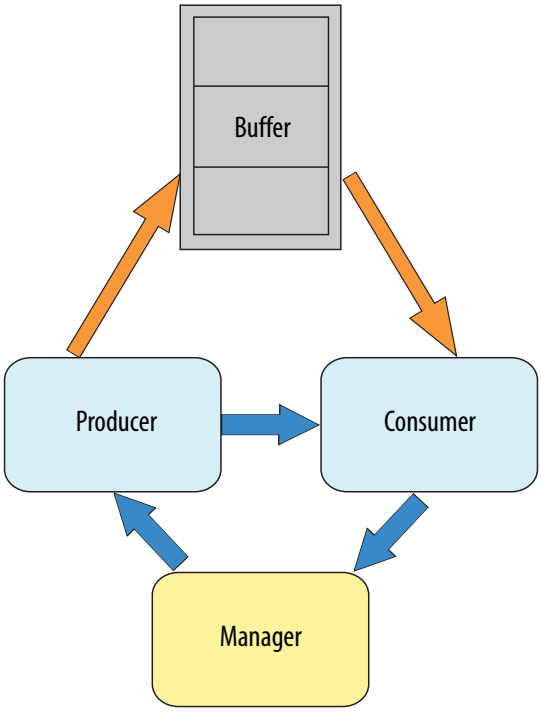
Figure 8. Feed-Forward Data Flow



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:

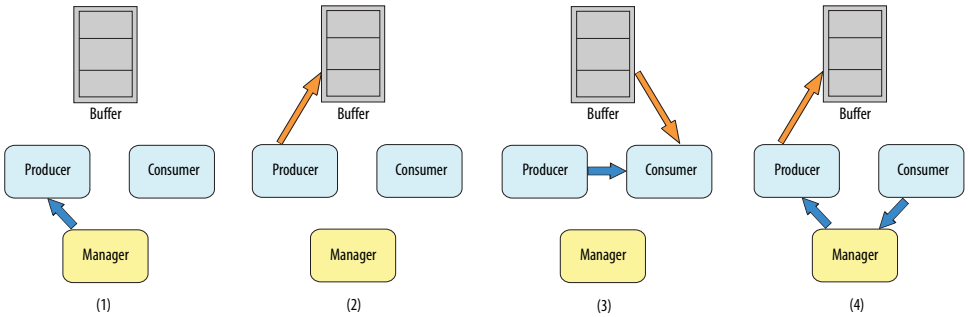
Figure 9. Feed-Forward Design Model with Buffer Management



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 10. Kernels Interaction during Buffer Management



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

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

Implementation of Buffer Management for OpenCL Kernels

To ensure that the SDK implements buffer management properly, the ordering of channel read and write operations is important. Consider the following kernel example:

```
__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_intel(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_intel(c, base_offset);
        }
    }
}
```

In this kernel, because the following lines of code are independent, the Intel FPGA SDK for OpenCL Offline Compiler can schedule them to execute concurrently:

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

and

```
write_channel_intel(c, base_offset);
```

Writing data to `base_offset` and then writing `base_offset` to a channel might be much faster than writing data to global memory. The `consumer` kernel might then read `base_offset` from the channel and use it as an index to read from global memory. Without synchronization, `consumer` might read data from `producer` before `shared_mem[base_offset + lID] = src[gID];` finishes executing. As a result, `consumer` reads in invalid data. To avoid this scenario, the synchronization token must occur after the `producer` kernel commits data to memory. In other words, a `consumer` kernel cannot consume data from the `producer` kernel until `producer` stores its data in global memory successfully.

To preserve this ordering, include an OpenCL `mem_fence` token in your kernels. 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_intel` call in the example cannot start until the global memory operation is committed to the shared memory buffer.

5.4.5.7. Implementing Buffered Channels Using the depth Channels Attribute

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

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

- If you expect any temporary mismatch between the consumption rate and the production rate to the channel, set the buffer size using the `depth` channel attribute.

  The following example demonstrates the use of the `depth` channel attribute in kernel code that implements the Intel FPGA SDK for OpenCL channels extension. The `depth(N)` attribute specifies the minimum depth of a buffered channel, where *N* is the number of data values.

```
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_intel(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_intel(c);
        }
        out_data[i] = last_val;
    }
}
```

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

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

5.4.5.8. Enforcing the Order of Channel Calls

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

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

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

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

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

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

- To prevent deadlocks from occurring by enforcing the order of channel calls, include memory fence functions (`mem_fence`) in your kernel.

  Inserting the `mem_fence` call with each kernel's channel flag forces the sequential ordering of the write and read channel calls. The code snippet below shows the modified `producer` and `consumer` kernels:

  ```
  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_intel(c0, src[2*i]);
          mem_fence(CLK_CHANNEL_MEM_FENCE);
          write_channel_intel(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_intel(c0);
          mem_fence(CLK_CHANNEL_MEM_FENCE);
  ```

```
        dst[2*i] = read_channel_intel(c1);
    }
}
```

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

5.4.5.8.1. Defining Memory Consistency Across Kernels When Using Channels

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

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

  For example:

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

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

5.5. Implementing OpenCL Pipes

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

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

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

The Intel FPGA SDK for OpenCL implementation of pipes does not encompass the entire pipes specification. As such, it is not fully conformant to the OpenCL Specification version 2.0. The goal of the SDK's pipes implementation is to provide a

solution that works seamlessly on a different OpenCL 2.0-conformant device. To enable pipes for Intel FPGA products, your design must satisfy certain additional requirements.

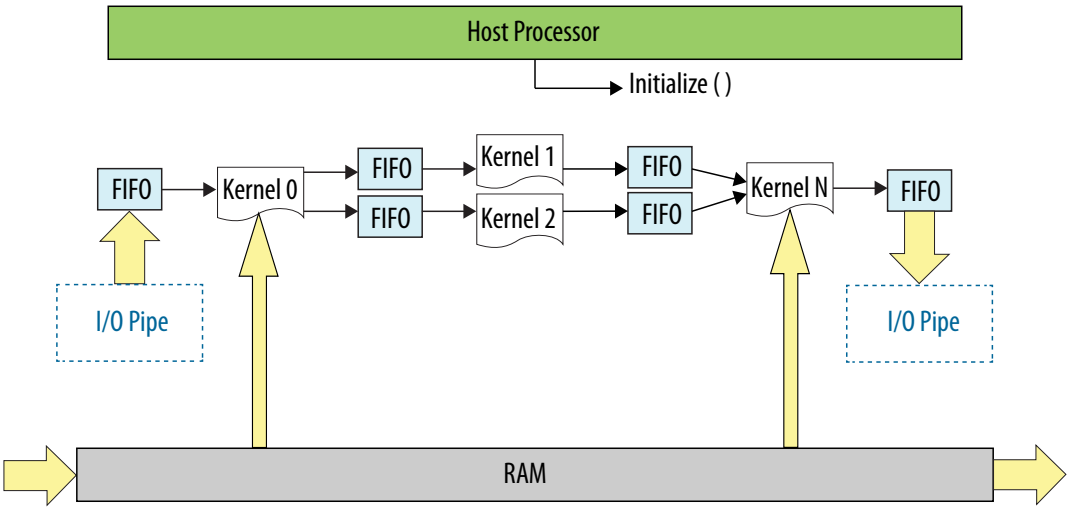
Related Information

OpenCL Specification version 2.0 (API)

5.5.1. Overview of the OpenCL Pipe Functions

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

Figure 11. Overview of a Pipe Network Implementation



Implementation of pipes decouples kernel execution from the host processor. The foundation of the Intel FPGA SDK for OpenCL pipes support is the SDK's channels extension. However, the syntax for pipe functions differs from the channels syntax.

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

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

Related Information

- Implementing Blocking Channel Writes on page 38
- Implementing Nonblocking Channel Writes on page 39
- Implementing Nonblocking Channel Reads on page 40
- Implementing Blocking Channel Reads on page 39

5.5.2. Pipe Data Behavior

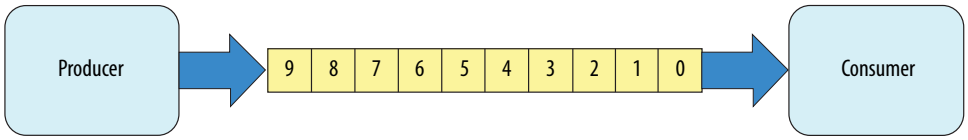
Data written to a pipe remains in a pipe as long as the kernel program remains loaded on the FPGA device. In other words, data written to a pipe persists across multiple work-groups and NDRange invocations. However, data is not persistent across multiple or different invocations of kernel programs that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

Consider the following code example:

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

__kernel void consumer (__global uint * restrict dst,
                        read_only pipe uint __attribute__((blocking))
                        __attribute__((depth(10))) c0)
{
    for (int i = 0; i < 5; i++)
    {
        read_pipe (c0, &dst[i]);
    }
}
```

A read operation to a pipe reads the *least* recent piece of data written to the pipe first. Pipe data maintains a FIFO ordering within the pipe.

Figure 12. Pipe Data FIFO Ordering



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

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

5.5.3. Multiple Work-Item Ordering for Pipes

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

Multiple work-item accesses to a pipe can be useful in some scenarios. For example, they are useful when data words in the pipe are independent, or when the pipe is implemented for control logic. The main concern regarding multiple work-item accesses to a pipe is the order in which the kernel writes data to and reads data from

the pipe. If possible, the OpenCL pipes process work-items read and write operations to a pipe in a deterministic order. As such, the read and write operations remain consistent across kernel invocations.

Requirements for Deterministic Multiple Work-Item Ordering

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

- All paths through the kernel must execute the pipe call.
- If the first requirement is not satisfied, none of the branch conditions that reach the pipe call should execute in a work-item-dependent manner.

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

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

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

5.5.3.1. Work-Item Serial Execution of Pipes

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

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

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

Pipe Execution in Loop with Multiple Work-Items

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

```
__kernel void ordering (__global int * data,
                        write_only pipe int __attribute__((blocking)) req)
{
    write_pipe (req, &data[get_global_id(0)]);
}
```

5.5.4. Restrictions in OpenCL Pipes Implementation

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

Default Behavior

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

Emulation Support

The Intel FPGA SDK for OpenCL Emulator supports emulation of kernels that contain pipes. The level of Emulator support aligns with the subset of OpenCL pipes support that is implemented for the FPGA hardware.

Pipes API Support

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

Single Call Site

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

```
read_pipe(pipe1, &in_data1);
read_pipe(pipe2, &in_data2);
read_pipe(pipe1, &in_data3);
```

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

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

```
read_pipe(pipe1, &in_data1);
read_pipe(pipe2, &in_data2);
read_pipe(pipe3, &in_data3);
```

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

```
#pragma unroll 4
for (int i = 0; i < 4; i++)
{
    read_pipe (pipe1, &in_data1);
}
```

The offline compiler issues the following warning message during compilation:

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

Feedback and Feed-Forward Pipes

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

Kernel Vectorization Support

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

Instruction-Level Parallelism on read_pipe and write_pipe Calls

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

Consider the following code sequence:

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

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

Related Information

OpenCL 2.0 C Programming Language Restrictions for Pipes on page 185

5.5.5. Enabling OpenCL Pipes for Kernels

To implement pipes, modify your OpenCL kernels to include pipes-specific API calls.

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

5.5.5.1. Ensuring Compatibility with Other OpenCL SDKs

Currently, Intel's implementation of OpenCL pipes is partially conformant to the OpenCL Specification version 2.0. If you port a kernel that implements pipes from another OpenCL SDK to the Intel FPGA SDK for OpenCL, you must modify the host code and the kernel code. The modifications do not affect subsequent portability of your application to other OpenCL SDKs.

Host Code Modification

Below is an example of a modified host application:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

    return ret;
}
```

Kernel Code Modification

If your kernel code runs on OpenCL SDKs that conforms to the OpenCL Specification version 2.0, you must modify it before running it on the Intel FPGA SDK for OpenCL. To modify the kernel code, perform the following modifications:

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

The modified kernel code appears as follows:

```
#define SIZE 1000

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

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

5.5.5.2. Declaring the Pipe Handle

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

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

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

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

  Consider the following pipe handle declarations:

  ```
  __kernel void first (pipe int c)

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

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

  Important: The Intel FPGA SDK for OpenCL Offline Compiler statically infers the connectivity of pipes in your system by matching the names of the pipe arguments. In the example above, the kernel `first` is connected to the kernel `second` by the pipe `c`.

In an Intel OpenCL system, only one kernel may read to a pipe. Similarly, only one kernel may write to a pipe. If a non-I/O pipe does not have at least one corresponding reading operation and one writing operation, the offline compiler issues an error.

For more information in the Intel FPGA SDK for OpenCL I/O pipe implementation, refer to *Implementing I/O Pipes Using the io Attribute*.

Related Information

Implementing I/O Pipes Using the io Attribute on page 61

5.5.5.3. Implementing Pipe Writes

The `write_pipe` API call allows you to send data across a pipe.

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

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

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

  Where:

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

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

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

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

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

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

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

Related Information

Profiling Your OpenCL Kernel on page 135

5.5.5.4. Implementing Pipe Reads

The `read_pipe` API call allows you to receive data across a pipe.

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

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

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

  Where:

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

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

  <type> defines the packet size of the data.

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

```
/*Declares the read_only_pipe that contains packets
of type long.*/
/*Declares that read_pipe calls within the kernel will exhibit
blocking behavior*/
__kernel void kernel_read_pipe(__global long *dst,
                               read_only pipe long __attribute__((blocking))
p)
{
```

```
    for (int i = 0; i < N; i++)
    {
        /*Reads from a long from the pipe and stores it
        into global memory at the specified location*/
        read_pipe(p, &dst[i]);
    }
}
```

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

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

5.5.5.5. Implementing Buffered Pipes Using the depth Attribute

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

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

In a buffered pipe, a write operation can only proceed if there is capacity in the pipe to hold the incoming packet. A read operation can only proceed if there is at least one packet in the pipe.

Use buffered pipes if pipe calls are predicated differently in the writer and reader kernels, and the kernels do not execute concurrently.

- If you expect any temporary mismatch between the consumption rate and the production rate to the pipe, set the buffer size using the `depth` attribute.

The following example demonstrates the use of the `depth` attribute in kernel code that implements the OpenCL pipes. The `depth(N)` attribute specifies the minimum depth of a buffered pipe, where *N* is the number of data values. If the read and write kernels specify different depths for a given buffered pipe, the Intel FPGA SDK for OpenCL Offline Compiler will use the larger depth of the two.

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

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

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

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

5.5.5.6. Implementing I/O Pipes Using the io Attribute

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

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

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

Caution:

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

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

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

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

The `width` attribute of an `interface` element specifies the width, in bits, of the data type used by that pipe. For the example above, both the `uint` and `float` data types are 32 bits wide. Other bigger or vectorized data types must match the appropriate bit width specified in the `board_spec.xml` file.

2. Implement the `io` attribute as demonstrated in the following code example. The `io` attribute names must match those of the I/O channels (`chan_id`) specified in the `board_spec.xml` file.

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

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

5.5.5.7. Enforcing the Order of Pipe Calls

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

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

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

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

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

- To prevent deadlocks from occurring by enforcing the order of pipe calls, include memory fence functions (`mem_fence`) in your kernel.

  Inserting the `mem_fence` call with each kernel's pipe flag forces the sequential ordering of the write and read pipe calls. The code snippet below shows the modified `producer` and `consumer` kernels:

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

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

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

5.5.5.7.1. Defining Memory Consistency Across Kernels When Using Pipes

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

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

  For example:

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

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

5.5.6. Direct Communication with Kernels via Host Pipes

The `cl_intel_fpga_host_pipe` extension enables point-to-point pipe communication between a kernel and the host program. Without the extension, pipes within OpenCL can only be used to communicate between kernels, and not with the host program directly.

The extension legalizes two new values in the `flags` argument of `clCreatePipe` to make a pipe host accessible, and adds four new API functions (`clReadPipeIntelFPGA`, `clWritePipeIntelFPGA`, `clMapHostPipeIntelFPGA`, `clUnmapHostPipeIntelFPGA`) to allow the host to read from and write to a pipe that was created with host access enabled. A new optional kernel argument attribute is added to specify in the kernel language that the opposing end of a pipe kernel argument will be the host program, and consequently that the pipe will not be connected to another kernel. A pipe kernel argument is specialized in the kernel definition to connect to either a host pipe or another kernel, and cannot dynamically switch between the two at runtime.

When a pipe kernel argument is marked for host accessibility, the kernel language pipe accessors are restricted to a subset of the 2.x functions (reservations are not supported), and memory consistency or visibility guarantees are made beyond OpenCL synchronization points.

Support for host accessible pipes is a device property, advertised as `cl_intel_fpga_host_pipe`.

Attention: A restriction of our implementation of host pipes is that the platform only supports two host pipes. One for read and one for write. Furthermore, the compiler accepts a pipe of only 32-bytes width, and hence `ulong4` is used in the *Example Use of `cl_intel_fpga_host_pipe` Extension* section.

Related Information

Example Use of the cl_intel_fpga_host_pipe Extension on page 67

5.5.6.1. Optional intel_host_accessible Kernel Argument Attribute

The `cl_intel_fpga_host_pipe` extension adds an optional `intel_host_accessible` kernel argument attribute. Applying this attribute to a kernel pipe argument specifies that the host will connect the kernel pipe argument to a host-accessible pipe, and not to another kernel pipe argument.

```
__attribute__((intel_host_accessible))
```

5.5.6.2. API Functions for Interacting with cl_mem Pipe Objects Bound to Host-Accessible Pipe Kernel Arguments

Including the `clReadPipeIntelFPGA`, `clWritePipeIntelFPGA`, `clMapHostPipeIntelFPGA`, and `clUnmapHostPipeIntelFPGA` host API functions allows the host program to read from and write to the `cl_mem` pipe objects that have been bound (using `clSetKernelArg` argument) to host-accessible pipe kernel arguments.

- `clReadPipeIntelFPGA` and `clWritePipeIntelFPGA` functions operate on single words of the pipe's width.
- `clMapHostPipeIntelFPGA` function is an advanced mechanism to reduce latency and overhead when performing many word reads or writes on a host pipe.
- `clUnmapHostPipeIntelFPGA` function allows the host program to signal to the OpenCL runtime that it has written to or read from either a portion of or the entire mapped region that was created through a previous `clMapHostPipeIntelFPGA` function call.

Table 1. API Functions for Bound cl_mem Objects

Function	Description
`cl_int clReadPipeIntelFPGA (cl_mem pipe, gentype *ptr);`	Reads a data packet from a pipe with the following characteristics: 1. Created with the `CL_MEM_HOST_READ_ONLY` flag 2. Bound to a kernel argument that has the `write_only` definition and the `intel_host_accessible` kernel argument attribute Each `clReadPipeIntelFPGA` function call reads one packet from the pipe. The operation is non-blocking; it does not wait until data is available in the pipe to successfully read before returning.
`cl_int clWritePipeIntelFPGA (cl_mem pipe, gentype *ptr);`	Writes a data packet to a pipe with the following characteristics: 1. Created with the `CL_MEM_HOST_WRITE_ONLY` flag 2. Bound to a kernel argument that has the `read_only` definition and the `intel_host_accessible` kernel argument attribute Each `clWritePipeIntelFPGA` function call writes one packet to the pipe. The operation is non-blocking; it does not wait until there is a capacity in the pipe to successfully write before returning. A return status of `CL_SUCCESS` does not imply that data is available to the kernel for reading. The data will eventually be available for reading by the kernel, assuming that any previously mapped buffers on the host pipe are unmapped.

continued...

Function	Description

`void * clMapHostPipeIntelFPGA (cl_mem pipe,`
`  cl_map_flags map_flags,`
`  size_t requested_size,`
`  size_t * mapped_size,`
`  cl_int * errcode_ret);`
	Returns a `void *` in the host address space. The pipe can write data to this address space if it was created with the `CL_MEM_HOST_WRITE_ONLY` flag. The pipe can read data from this address space if it was created with the `CL_MEM_HOST_READ_ONLY` flag.
	The `mapped_size` argument specifies the maximum number of bytes that the host can access, as determined by the runtime in the memory. The value specified by mapped_size is less than or equal to the value of the `requested_size` argument that the caller specifies.
	After writing to or reading from the returned `void *`, the host must execute one or more `clUnmapHostPipeIntelFPGA` function calls to signal to the runtime that data is ready for transfer to the device (on a write), and that the runtime can reclaim the memory for reuse (on a read or write). If the `clMapHostPipeIntelFPGA` function is called before the `clUnmapHostPipeIntelFPGA` function unmaps all memory mapped by a previous `clMapHostPipeIntelFPGA` function call, the buffer returned by the second `clMapHostPipeIntelFPGA` call will not overlap with that returned by the first call.

`cl_int clUnmapHostPipeIntelFPGA (`
`  cl_mem pipe,`
`  void * mapped_ptr,`
`  size_t size_to_unmap,`
`  size_t * unmapped_size`
`  );`
	Signals to the runtime that the hostno longer uses `size_to_unmap` bytes of a host-addressable buffer that the `clMapHostPipeIntelFPGA` function has returned previously. In the case of a writeable host pipe, calling `clUnmapHostPipeIntelFPGA` allows the unmapped data to become available to the kernel. If the `size_to_unmap` value is smaller than the `mapped_size` value specified by the `clMapHostPipeIntelFPGA` function, then multiple `clUnmapHostPipeIntelFPGA` function calls are necessary to unmap the full capacity of the buffer. You may include multiple `clUnmapHostPipeIntelFPGA` function calls to unmap successive bytes in the buffer returned by a `clMapHostPipeIntelFPGA` function call, up to the `mapped_size` value defined by the `clMapHostPipeIntelFPGA` call.

5.5.6.3. Creating a Host Accessible Pipe

The `clCreatePipe` function, defined in the *Section 5.4.1* of the *OpenCL 2.2 API Specification*, contains a `flags` parameter. The legal values of `flags` for `clCreatePipe` function are `CL_MEM_READ_WRITE` and `CL_MEM_HOST_NO_ACCESS`. If the value passed to `flags` is 0, then the specification defines that both of these flags are implicitly passed as the default.

To enable host access (reading or writing) to pipes, the `cl_intel_fpga_host_pipe` extension legalizes the following two `flags` values to `clCreatePipe`:

- `CL_MEM_HOST_READ_ONLY`
- `CL_MEM_HOST_WRITE_ONLY`

When one of these `flags` is passed to the `clCreatePipe` function, the corresponding `cl_mem` object can be passed as the first argument to `clReadPipeIntelFPGA` and `clWritePipeIntelFPGA` functions. Throughout the remainder of the `cl_intel_fpga_host_pipe` extension, such a pipe is referred to as a host pipe.

Warning: It is illegal to specify both `CL_MEM_HOST_READ_ONLY` and `CL_MEM_HOST_WRITE_ONLY` on the same pipe, or to mix either of those values with `CL_MEM_READ_WRITE` or `CL_MEM_HOST_NO_ACCESS`, or both. Invalid `flags` combinations will be detected by the OpenCL runtime, and will cause `clCreatePipe` to return the `CL_INVALID_VALUE` error.

5.5.6.4. Example Use of the cl_intel_fpga_host_pipe Extension

The following are the example kernel and host codes of the `cl_intel_fpga_host_pipe` extension:

Kernel Code

```
#pragma OPENCL EXTENSION cl_intel_fpga_host_pipe : enable

kernel void reader(__attribute__((intel_host_accessible))
                   __read_only pipe ulong4 host_in) {
  ulong4 val;
  if (read_pipe(host_in, &val)) {
    ....
  }
  ....
}

kernel void writer(__attribute__((intel_host_accessible))
                   __write_only pipe ulong4 device_out) {
  ulong4 val;
  ....
  if (write_pipe(device_out, &val)) {
    ....
  }
}
```

Host Code

```
....

cl_kernel read_kern = clCreateKernel(program, "reader", NULL);

cl_kernel write_kern = clCreateKernel(program, "writer", NULL);

cl_mem read_pipe = clCreatePipe(context, CL_MEM_HOST_READ_ONLY,
                                sizeof( cl_ulong4 ), 128,
                                // Number of packets that can be buffered
                                NULL, &error);

cl_mem write_pipe = clCreatePipe(context, CL_MEM_HOST_WRITE_ONLY,
                                 sizeof( cl_ulong4 ), 64,
                                 // Number of packets that can be buffered
                                 NULL, &error);

// Bind pipes to kernels
clSetKernelArg(read_kern, 0, sizeof(cl_mem), (void *)&write_pipe);
clSetKernelArg(write_kern, 0, sizeof(cl_mem), (void *)&read_pipe);

// Enqueue kernels
....

cl_ulong4 val
if (!clReadPipeIntelFPGA (read_pipe, &val)) {
  cl_int result = clWritePipeIntelFPGA (write_pipe, &val);
  // Check write success/failure and handle
  ....
}

....
```

5.6. Implementing Arbitrary Precision Integers

Use the Intel FPGA SDK for OpenCL arbitrary precision integer extension to define integers with a custom bit-width. You can define integer custom bit-widths up to and including 64 bits.

To use the arbitrary precision integer extension, include the following line in your list of header files in your kernel code:

```
#include "ihc_apint.h"
```

When you compile a kernel that includes the `ihc_apint.h` header file, you must include the `-I $INTELFPGAOCLSDKROOT/include/kernel_headers` option with the `aoc` command. For example:

```
aoc <other command options> -I $INTELFPGAOCLSDKROOT/include/kernel_headers
<my_kernel_file>
```

The header enables the arbitrary precision integers extension, and has macros that define C-style declarations for signed and unsigned arbitrary precision integers as follows:

```
#define ap_int<d> intd_t
#define ap_uint<d> uintd_t
```

For example, you can declare a 10-bit signed and unsigned arbitrary precision integers as follows:

```
int10_t x_signed;
uint10_t x_unsigned;
```

You can declare arbitrary precision integers with widths up to 64 bits.

To use arbitrary precision integers without using the header files, enable the extension with the following pragma directive:

```
#pragma OPENCL EXTENSION cl_intel_arbitrary_precision_integers : enable
```

After the pragma declaration, you can declare your arbitrary precision integers as follows:

```
ap_int<d> intd_t my_signed_integer
ap_uint<d> uintd_t my_unsigned_integer
```

If you do operations where the bit width of the result is larger than the bit widths of the arguments, you must explicitly cast one of the arguments to the resulting bit width.

For example, if you had the following operation, the result overflows the declared size of the integer:

```
int10_t a;
int10_t b;
int20_t res;

res = a * b;
```

In the example, the compiler attempts to instantiate a multiplier that multiplies two 10-bit integers and put the results into another 10-bit integer. The result is then sign extended or zero extended up to 20-bits.

To prevent the overflow, explicitly cast one of the arguments to the resulting bit width as follows:

```
res = ((int20_t)a) * b
```

Remember: When you compile a program for x86-64 platforms, the bit widths for arbitrary precisions integers are rounded up to either 32 bits or 64 bits. When you compile a kernel for an FPGA platform, the bit widths are not rounded up and the arbitrary precision integers remain at their declared bit width.

As a result, an operation that appears to work correctly in an x86-64 program can overflow and lose precision when you compile that same operation in an FPGA kernel. The additional precision provided by bit-width rounding on x86-64 platforms masks possible overflow and precision-loss problems you might encounter when your compile your FPGA kernel.

5.7. Using Predefined Preprocessor Macros in Conditional Compilation

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

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

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

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

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

  Where INTELFPGA_CL is the Intel predefined preprocessor macro for the offline compiler.

Related Information

Defining Preprocessor Macros to Specify Kernel Parameters (-D<macro_name>) on page 109

5.8. Declaring __constant Address Space Qualifiers

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

Function Scope __constant Variables

The Intel FPGA SDK for OpenCL Offline Compiler does not support function scope `__constant` variables. Replace function scope `__constant` variables with file scope constant variables. You can also replace function scope `__constant` variables with `__constant` buffers that the host passes to the kernel.

File Scope __constant Variables

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

For example:

```
__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 offline compiler sets the values for `my_array` in a ROM because the file scope constant data does not change between kernel invocations.

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 if the data is not fixed across kernel invocations. 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 2. 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.

5.9. Including Structure Data Types as Arguments in OpenCL Kernels

Pass structure parameters (`struct`) in OpenCL kernels either by value or as a pointer to a structure.

Attention: The `__global struct` argument points to a buffer that must be created in the host program to store the structure. To prevent pointer aliasing, include a `restrict` qualifier in the declaration of the pointer to the structure.

5.9.1. Matching Data Layouts of Host and Kernel Structure Data Types

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

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

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

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

The following rules apply when the Intel FPGA SDK for OpenCL Offline Compiler 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 offline compiler usually aligns a data type based on its size. However, the compiler 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 offline compiler 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 offline compiler does not reorder data members of a `struct`.
5. Normally, the offline compiler inserts a minimum amount of data structure padding between data members of a `struct` to satisfy the alignment requirements for each data member.
   a. In your OpenCL kernel code, you may specify data packing (that is, no insertion of data structure padding) by applying the `packed` attribute to the `struct` declaration. If you impose data packing, ensure that the alignment of data members satisfies the OpenCL alignment requirements. The Intel FPGA SDK for OpenCL does not enforce these alignment requirements. Ensure that your host compiler respects the kernel attribute and sets the appropriate alignments.
   b. In your OpenCL kernel code, you may specify the amount of data structure padding by applying the `aligned(N)` attribute to a data member, where *N* is the amount of padding. The SDK does not enforce these alignment requirements. Ensure that your host compiler respects the kernel attribute and sets the appropriate alignments.

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

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

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

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

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

Related Information

- Modifying Host Program for Structure Parameter Conversion on page 89
- OpenCL Specification version 1.0

5.9.2. Disabling Insertion of Data Structure Padding

You may instruct the Intel FPGA SDK for OpenCL Offline Compiler to disable automatic padding insertion between members of a `struct` data structure.

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

  For example:

```
struct __attribute__((packed)) Context
{
   float param1;
   float param2;
```

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

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

Related Information

Align a Struct with or without Padding

5.9.3. Specifying the Alignment of a Struct

You may instruct the Intel FPGA SDK for OpenCL Offline Compiler to set a specific alignment of a `struct` data structure.

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

  For example:

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

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

Related Information

Align a Struct with or without Padding

5.10. Inferring a Register

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

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

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

Consider the following code example:

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

The indexing into `array[i]` is not inferable statically because the loop is not unrolled. If the size of `array[SIZE]` is less than or equal to 64 bytes for single work-item kernels, the offline compiler implements `array[SIZE]` into registers as a single value. If the size of `array[SIZE]` is greater than 64 bytes for single work-item kernels, the offline compiler implements the entire array in block RAMs. For multiple work-item kernels, the offline compiler implements `array[SIZE]` into registers as a single value as long as its size is less than 1 kilobyte (KB).

5.10.1. Inferring a Shift Register

The shift register design pattern is a very important design pattern for efficient implementation of many applications on the FPGA. 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_intel(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_intel(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 Intel FPGA SDK for OpenCL Offline Compiler can efficiently handle multiple access points into the array. The shift register design pattern is ideal for implementing filters (for example, image filters like a Sobel filter or time-delay filters like a finite impulse response (FIR) filter).

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

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

5.11. Enabling Double Precision Floating-Point Operations

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

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

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

5.12. Single-Cycle Floating-Point Accumulator for Single Work-Item Kernels

Single work-item kernels that perform accumulation in a loop can leverage the single-cycle floating-point accumulator feature of the Intel FPGA SDK for OpenCL Offline Compiler. The offline compiler searches for these kernel instances and attempts to map an accumulation that executes in a loop into the accumulator structure.

The offline compiler supports an accumulator that adds or subtracts a value. To leverage this feature, describe the accumulation in a way that allows the offline compiler to infer the accumulator.

Attention:

- The accumulator is only available on Arria 10 devices.
- The accumulator must be part of a loop.
- The accumulator must have an initial value of 0.
- The accumulator cannot be conditional.

Below are examples of a description that results in the correct inference of the accumulator by the offline compiler.

```
channel float4 RANDOM_STREAM;

__kernel void acc_test(__global float *a, int k) {
    // Simplest example of an accumulator.
    // In this loop, the accumulator acc is incremented by 5.
    int i;
    float acc = 0.0f;
    for (i = 0; i < k; i++) {
        acc+=5;
    }
    a[0] = acc;
}

__kernel void acc_test2(__global float *a, int k) {
    // Extended example showing that an accumulator can be
    // conditionally incremented. The key here is to describe the increment
    // as conditional, not the accumulation itself.
    int i;
    float acc = 0.0f;
    for (i = 0; i < k; i++) {
        acc += ((i < 30) ? 5 : 0);
    }
    a[0] = acc;
}

__kernel void acc_test3(__global float *a, int k) {
    // A more complex case where the accumulator is fed
    // by a dot product.
    int i;
    float acc = 0.0f;
    for (i = 0; i < k; i++ ){
        float4 v = read_channel_intel(RANDOM_STREAM);
        float x1 = v.x;
        float x2 = v.y;
        float y1 = v.z;
        float y2 = v.w;

        acc += (x1*y1+x2*y2);
    }
    a[0] = acc;
}

__kernel void loader(__global float *a, int k) {
    int i;
    float4 my_val = 0;
    for(i = 0; i < k; i++) {
        if ((i%4) == 0)
            write_channel_intel(RANDOM_STREAM, my_val);
        if ((i%4) == 0) my_val.x = a[i];
        if ((i%4) == 1) my_val.y = a[i];
        if ((i%4) == 2) my_val.z = a[i];
        if ((i%4) == 3) my_val.w = a[i];
    }
}
```

5.12.1. Programming Strategies for Inferring the Accumulator

To leverage the single cycle floating-point accumulator feature, you can modify the accumulator description in your kernel code to improve efficiency or work around programming restrictions.

Describing an Accumulator Using Multiple Loops

Consider a case where you want to describe an accumulator using multiple loops, with some of the loops being unrolled:

```
float acc = 0.0f;
for (i = 0; i < k; i++) {
    #pragma unroll
    for(j=0;j < 16; j++)
        acc += (x[i+j]*y[i+j]);
}
```

In this situation, it is important to compile the kernel with the `-fp-relaxed` Intel FPGA SDK for OpenCL Offline Compiler command option to enable the offline compiler to rearrange the operations in a way that exposes the accumulation. If you do not compile the kernel with `-fp-relaxed`, the resulting accumulator structure will have a high initiation interval (II). II is the number of cycles between launching successive loop iterations. The higher the II value, the longer the accumulator structure must wait before it can process the next loop iteration.

Modifying a Multi-Loop Accumulator Description

In cases where you cannot compile an accumulator description using the `-fp-relaxed` offline compiler command option, rewrite the code to expose the accumulation.

For the code example above, rewrite it in the following manner:

```
float acc = 0.0f;
for (i = 0; i < k; i++) {
    float my_dot = 0.0f;
    #pragma unroll
    for(j=0;j < 16; j++)
        my_dot += (x[i+j]*y[i+j]);
    acc += my_dot;
}
```

Modifying an Accumulator Description Containing a Variable or Non-Zero Initial Value

Consider a situation where you might want to apply an offset to a description of an accumulator that begins with a non-zero value:

```
float acc = array[0];
for (i = 0; i < k; i++) {
    acc += x[i];
}
```

Because the accumulator hardware does not support variable or non-zero initial values in a description, you must rewrite the description.

```
float acc = 0.0f;
for (i = 0; i < k; i++) {
    acc += x[i];
}
acc += array[0];
```

Rewriting the description in the above manner enables the kernel to use an accumulator in a loop. The loop structure is then followed by an increment of `array[0]`.

6. Designing Your Host Application

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

6.1. Host Programming Requirements

When designing your OpenCL host application for use with the Intel FPGA SDK for OpenCL, ensure that the application satisfies the following host programming requirements.

6.1.1. 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.

6.1.2. Host Binary Requirement

When compiling the host application, target one of these architectures: x86-64 (64-bit) or ARM® 32-bit ARMV7-A for SoCs. The Intel FPGA SDK for OpenCL host runtime does not support x86-32 (32-bit) binaries.

6.1.3. Multiple Host Threads

The Intel FPGA SDK for OpenCL host library is thread-safe.

All OpenCL APIs are thread safe except the `clSetKernelArg` function.

It is safe to call `clSetKernelArg` from any host thread or in a reentrant way as long as concurrent calls to `clSetKernelArg` operate on different `cl_kernel` objects.

Related Information

Multi-Threaded Host Application

6.1.4. Out-of-Order Command Queues

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

6.1.5. Requirement for Multiple Command Queues to Execute Kernels Concurrently

To execute kernels within the same OpenCL program object concurrently, instantiate a separate command for each kernel you wish to run concurrently.

6.2. Allocating OpenCL Buffers for Manual Partitioning of Global Memory

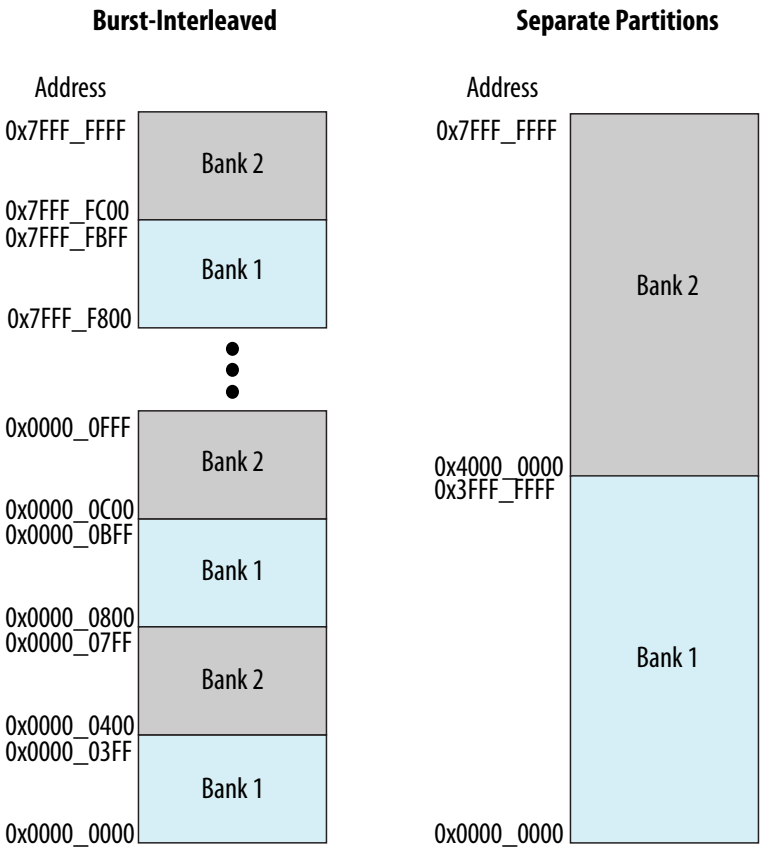
Manual partitioning of global memory buffers allows you to control memory accesses across buffers to maximize the memory bandwidth. You can partition buffers across interfaces of the same memory type or across interfaces of different memory types.

6.2.1. Partitioning Buffers Across Multiple Interfaces of the Same Memory Type

Before you partition the memory across multiple interfaces of the same memory type, you must first disable burst-interleaving during OpenCL kernel compilation. Then, in the host application, you must specify the memory bank to which you allocate the OpenCL buffer.

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

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



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

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

   For more information on the usage of the `-no-interleaving=<global_memory_type>` flag, refer to the *Disabling Burst-Interleaving of Global Memory (-no-interleaving=<global_memory_type>)* section.

2. Create an OpenCL buffer in your host application, and allocate the buffer to one of the banks using the `CL_CHANNEL` flags.

   — Specify `CL_CHANNEL_1_INTELFPGA` to allocate the buffer to the lowest available memory region.

   — Specify `CL_CHANNEL_2_INTELFPGA` to allocation memory to the second bank (if available).

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

Disabling Burst-Interleaving of Global Memory (-no-interleaving=<global_memory_type>) on page 113

6.2.2. Partitioning Buffers Across Different Memory Types (Heterogeneous Memory)

The board support package for your FPGA board can assemble a global memory space consisting of different memory technologies (for example, DRAM or SRAM). The board support package designates one such memory, which might consist of multiple interfaces, as the default memory. All buffers reside there.

To use the heterogeneous memory, modify the code in your `.cl` file as follows:

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

   For example:

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

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

   Intel recommends that you define the `buffer_location` attribute in a preprocessor macro for ease of reuse, as follows:

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

   #define DDR\
   __global __attribute__((buffer_location("DDR")))
   __kernel void foo (QDR uint * data, DDR uint * lup)
   {
       //statements
   }
   ```

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

By default, the host allocates buffers into the main memory when you load kernels into the OpenCL runtime via the `clCreateProgramWithBinary` function. During kernel invocation, the host automatically relocates heterogeneous memory buffers that are bound to kernel arguments to the main memory.

3. To avoid the initial allocation of heterogeneous memory buffers in the main memory, include the `CL_MEM_HETEROGENEOUS_INTELFPGA` flag when you call the `clCreateBuffer` function. Also, bind the `cl_mem` buffer to the argument that used the `buffer_location` attribute using `clSetKernelArg` before doing any reads or writes from that buffer, as follows:

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

clSetKernelArg(kernel, 0, sizeof(cl_mem), &mem);
clEnqueueWriteBuffer(queue, mem, CL_FALSE, 0, N, 0, NULL, &write_event);
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, global_work_size, NULL,
                       0, NULL, &kernel_event);
```

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

```
mem = clCreateBuffer(context,
                (CL_MEM_HETEROGENEOUS_INTELFPGA|CL_CHANNEL_1_INTELFPGA),
                 memSize,
                 NULL,
                 &errNum);
```

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

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 *Intel FPGA SDK for OpenCL Best Practices Guide*.

Related Information

- Manual Partitioning of Global Memory
- Heterogeneous Memory Buffers

6.2.3. Creating a Pipe Object in Your Host Application

To implement OpenCL pipes in your kernel, you must create Intel FPGA SDK for OpenCL-specific pipe objects in your host application.

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

The following `clCreatePipe` host API creates a pipe object:

```
cl_mem clCreatePipe(cl_context context,
                    cl_mem_flags flags,
                    cl_uint pipe_packet_size,
```

```
                    cl_uint pipe_max_packets,
                    const cl_pipe_properties *properties,
                    cl_int *errcode_ret)
```

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

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

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

Caution: The SDK does not support dynamic channel assignment at runtime. The SDK statically links the pipes during compilation based on matching names.

Related Information

OpenCL Specification version 2.0 (API)

6.3. 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. The default behavior of automatic readback of profile data upon the completion of kernel execution is sufficient for most applications.

When you profile your OpenCL kernel during compilation, a `profile.mon` file is generated automatically. The profile data is then written to `profile.mon` after kernel execution completes on the FPGA. However, if kernel execution completes after the host application terminates, 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.

Important: Collecting profile data during kernel execution can add significant overhead to kernel executions by increasing the latency in your kernel.

- 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

clGetProfileInfoIntelFPGA(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 `clGetProfileInfoIntelFPGA`, the function returns an event error message.

  Caution: Invoking the `clGetProfileInfoIntelFPGA` function during kernel execution disables the profile counters momentarily so that the Intel FPGA Dynamic Profiler for OpenCL 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
    clGetProfileInfoIntelFPGA(event);

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

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

  The call to `clGetProfileInfoIntelFPGA` adds a new entry in the `profile.mon` file. The Intel FPGA Dynamic Profiler for OpenCL GUI then parses this entry in the report.

For more information on the Intel FPGA Dynamic Profiler for OpenCL, refer to the following sections:

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

Related Information

- Profile Your Kernel to Identify Performance Bottlenecks
- Profiling Your OpenCL Kernel on page 135

6.3.1. Profiling Enqueued and Autorun Kernels

Unlike enqueued kernels that automatically generate profiler data on completion (if the compiler flag is set), autorun kernels never complete. Hence, you must explicitly indicate when to profile kernels by calling the `clGetProfileDataDeviceIntelFPGA` host library call. All profiler data is output to a `profile.mon` file. Data collected by the host library call is a snapshot of the autorun profile data.

Following is the code snippet of the `clGetProfileDataDeviceIntelFPGA` host library call:

```
cl_int clGetProfileDataDeviceIntelFPGA (cl_device_id device_id,
                                        cl_program program,
                                        cl_bool read_enqueue_kernels,
                                        cl_bool read_auto_enqueued,
                                        cl_bool clear_counters_after_readback,
                                        size_t param_value_size,
                                        void *param_value,
                                        size_t *param_value_size_ret,
                                        cl_int *errcode_ret);
```

where,

- `read_enqueue_kernels` parameter profiles enqueued kernels. In this release, this parameter has no effect.
- `read_auto_enqueued` parameter profiles autorun kernels.
- Following are the placeholder parameters for the future releases:
  - `clear_counters_after_readback`
  - `param_value_size`
  - `param_value`
  - `param_value_size_ret`
  - `errcode_ret`

Notice: Only autorun kernels are supported by this host library call. You can enter TRUE for the `read_enqueue_kernels` parameter, but the boolean is ignored. This does not mean that enqueued kernels are not profiled. If the compiler `profile` flag is set to include enqueued kernels, the profile data is captured normally at the end of execution. The only difference is that the `clGetProfileDataDeviceIntelFPGA` host library call does not profile enqueued kernels in addition to the profiling already done automatically for the enqueued kernels.

The `clGetProfileDataDeviceIntelFPGA` host library call returns `CL_SUCCESS` on success. Else, it returns one of the following errors:

- `CL_INVALID_DEVICE` if the device is not a valid device.
- `CL_INVALID_PROGRAM` if the program is not a valid program.

Caution: The `clGetProfileDataDeviceIntelFPGA` host library call will not trigger a programming operation of the provided program on the provided device. If the program is not already programmed to the device at the time of the host library call, then the host library call returns `CL_INVALID_PROGRAM` error.

Table 3. `clGetProfileDataDeviceIntelFPGA` Host Library Call Parameter Combinations

	read_auto_enqueued
Profile only enqueued kernels *Note:* Automatically outputs profile information once the execution is completed.	False
Profile only autorun kernels	True
Profile both enqueued and autorun kernels	True

6.3.2. Profile Data Acquisition

Profile data acquisition from running kernels is paused during read back operations.

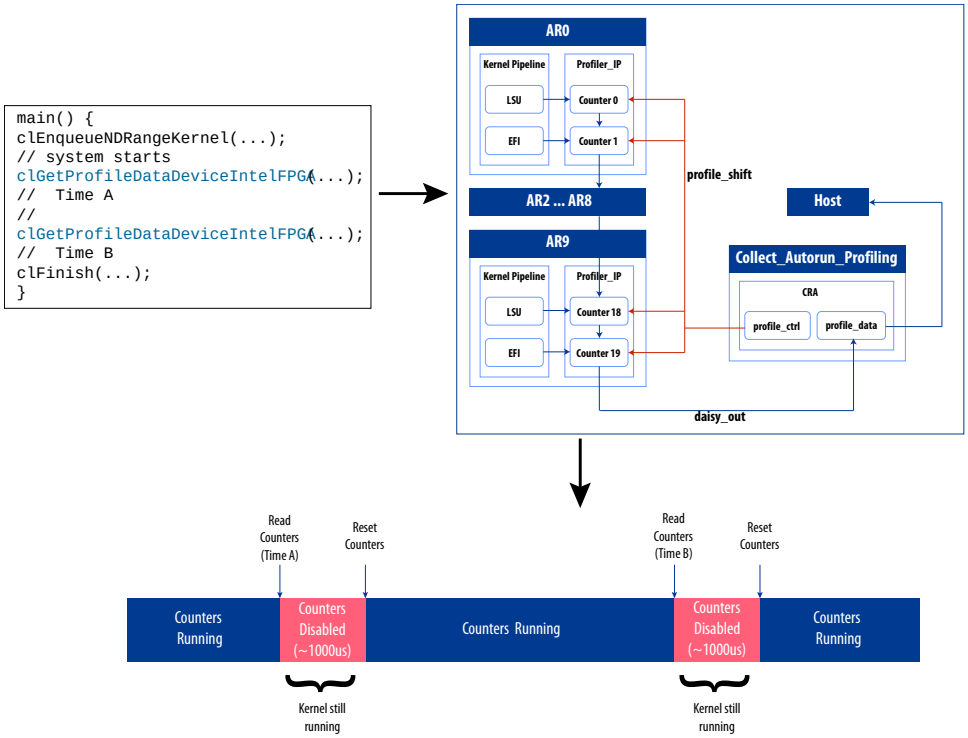
Attention: Although data acquisition is paused, kernels themselves are still running. Therefore, during read back operations, no kernel data is recorded.

Pausing data acquisition is not synchronized exactly across all kernels. The skew between halting profile data acquisition across kernels is dependent on the communication link with the device, driver overhead, and congestion on communication buses. Exact synchronized snapshotting of profile data between kernels should not be relied upon.

6.3.3. Multiple Autorun Profiling Calls

Because autorun kernels run continuously, the host application can include multiple `clGetProfileDataDeviceIntelFPGA` host library calls to profile autorun kernels at certain points within the execution or in specific time ranges. Every time the host application calls the `clGetProfileDataDeviceIntelFPGA` host library call, it reads the profile counters and then resets them to zero. Calling `clGetProfileDataDeviceIntelFPGA` multiple times allows the host application to profile autorun kernels over time ranges.

Figure 13. Multiple Autorun Profile Captures Flow



6.4. Accessing Custom Platform-Specific Functions

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

The `clGetBoardExtensionFunctionAddressIntelFPGA` extension specifies an API that retrieves a pointer to a user-accessible function from the Custom Platform.

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

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

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

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

Where:

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

and

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

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

Attention: To access the `clGetBoardExtensionFunctionAddressIntelFPGA` API via the Installable Client Driver (ICD), ensure that the ICD extension API `clGetExtensionFunctionAddressIntelFPGA` retrieves the pointer to the `clGetBoardExtensionFunctionAddressIntelFPGA` API first.

The following code example shows how you can access the Custom Platform-specific function via ICD:

```
clGetBoardExtensionFunctionAddressIntelFPGA_fn
   clGetBoardExtensionFunctionAddressIntelFPGA =
      (clGetBoardExtensionFunctionAddressIntelFPGA_fn)
   clGetExtensionFunctionAddressForPlatform
      (platform, "clGetBoardExtensionFunctionAddressIntelFPGA");

if (clGetBoardExtensionFunctionAddressIntelFPGA == NULL){
   printf ("Failed to get
           clGetBoardExtensionFunctionAddressIntelFPGA\n");
}

void * board_extension_function_ptr =

clGetBoardExtensionFunctionAddressIntelFPGA("function_name",device_id);
```

Related Information

- Linking Your Host Application to the Khronos ICD Loader Library on page 94
- OpenCL Installable Client Driver (ICD) Loader

6.5. Modifying Host Program for Structure Parameter Conversion

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

Perform the following changes to your host application:

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

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

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

Related Information

- Including Structure Data Types as Arguments in OpenCL Kernels on page 71
- Matching Data Layouts of Host and Kernel Structure Data Types on page 71

6.6. Managing Host Application

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

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

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

```
handle SIG44 nostop
```

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

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

6.6.1. Displaying Example Makefile Fragments (example-makefile or makefile)

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

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

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


Example GNU makefile on Linux, with GCC toolchain:

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

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

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


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

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

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

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


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

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

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

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

6.6.2. Compiling and Linking Your Host Application

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

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

Displaying Flags for Compiling Host Application (compile-config) on page 92
Displaying Paths to OpenCL Host Runtime and MMD Libraries (ldflags) on page 92
Listing OpenCL Host Runtime and MMD Libraries (ldlibs) on page 92
Displaying Information on OpenCL Host Runtime and MMD Libraries (link-config or linkflags) on page 93

6.6.2.1. Displaying Flags for Compiling Host Application (compile-config)

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

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

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

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

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

To display the paths necessary for linking a host application to the OpenCL host runtime and MMD libraries, invoke the `ldflags` utility command.

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

Note: If you set up FCD correctly, the software will not print the path to the MMD libraries because the host no longer needs to link to the MMD libraries directly. The MMD libraries will be loaded during runtime through FCD.

6.6.2.3. Listing OpenCL Host Runtime and MMD Libraries (ldlibs)

To display the names of the OpenCL host runtime and MMD libraries necessary for linking a host application, invoke the `ldlibs` utility command.

- At a command prompt, invoke the `aocl ldlibs` utility command.

  The software lists the OpenCL host runtime libraries residing in the *INTELFPGAOCLSDKROOT*`/host/`*<OS_platform>*`/lib` directory. It also lists the Custom Platform-specific MMD libraries residing in the */<board_family_name>/<OS_platform>*`/lib` directory of your Custom Platform.

  Note: If you set up FCD correctly, the software will not list the MMD libraries.

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

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

  If you set up FCD correctly, the output will be `OpenCL.lib`.

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

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

  If you set up FCD correctly, the output will be `-lOpenCL`.

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

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

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

1. At a command prompt, invoke the `aocl link-config` or `aocl linkflags` command.
   The software displays the link options for linking your host application with the following libraries:
   a. The path to and the names of OpenCL host runtime libraries that provide OpenCL platform and runtime APIs. The OpenCL host runtime libraries are available in the *INTELFPGAOCLSDKROOT*`/host/`*<OS_platform>*`/lib` directory .
   b. The path to and the names of the Custom Platform-specific MMD libraries. The MMD libraries are available in the */<board_family_name>/<OS_platform>*`/lib` directory of your Custom Platform.

Note: If you set up FCD correctly, the software will not print the path and names of the MMD libraries because the host no longer needs to link to the MMD libraries directly. The MMD libraries will be loaded during runtime through the FCD loader.

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

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

If you set up FCD correctly, the output will be `/libpath:%INTELFPGAOCLSDKROOT%/host/windows64/lib OpenCL.lib`

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

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

If you set up FCD correctly, the output will be `-L/$INTELFPGAOCLSDKROOT/host/[linux64|arm32]/lib -lOpenCL`

6.6.3. Linking Your Host Application to the Khronos ICD Loader Library

The Intel FPGA SDK for OpenCL supports the OpenCL ICD extension from the Khronos Group. The OpenCL ICD extension allows you to have multiple OpenCL implementations on your system. With the OpenCL ICD Loader Library, you may choose from a list of installed platforms and execute OpenCL API calls that are specific to your OpenCL implementation of choice.

In addition to the SDK's host runtime libraries, Intel supplies a version of the ICD Loader Library that supports the OpenCL Specification version 1.0 and the implemented APIs from the OpenCL Specification versions 1.1, 1.2, and 2.0. To use an ICD library from another vendor, consult the vendor's documentation on how to link to their ICD library.

Linking to the ICD Loader Library on Windows on page 95

Linking to the ICD Loader Library on Linux on page 95

6.6.3.1. Linking to the ICD Loader Library on Windows

Before linking your Windows OpenCL host application to the ICD Loader Library, ensure that you have set up FCD correctly. The Intel FPGA SDK for OpenCL utility `aocl link-config` will then automatically output the required flags for linking to the ICD Loader Library.

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

- If you need to manually set up FCD support for your Custom Platform, perform the following tasks:
  a. Consult with your board vendor to identify the libraries that the FCD requires. Alternatively, you can invoke the `aocl ldlibs` command and identify the libraries that your OpenCL application requires.
  b. Specify the libraries in the registry key **HKEY_LOCAL_MACHINE \SOFTWARE\Intel\OpenCL\Boards**. Specify the value name to be the path to the library, and specify the data to be a **DWORD** that is set to 0.

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

To enumerate board vendor-specific libraries, FCD scans the values in the **HKEY_LOCAL_MACHINE\SOFTWARE\Intel\OpenCL\Boards** registry key. For each **DWORD** value that is set to 0, the FCD Loader opens the corresponding DLL that is specified in the value name.

Consider the following registry key value:

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

The FCD loader scans this value and loads the library `my_board_mmd.dll` from the `board_vendor a` folder.

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

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

6.6.3.2. Linking to the ICD Loader Library on Linux

Before linking your Linux OpenCL host application to the ICD Loader Library, ensure that you have set up FCD correctly. The Intel FPGA SDK for OpenCL utility `aocl link-config` will then automatically output the required flags for linking to the ICD Loader Library.

- If you need to manually set up FCD support for your Custom Platform, perform the following tasks:
  a. Consult with your board vendor to identify the libraries that the FCD requires. Alternatively, you can invoke the `aocl ldlibs` command and identify the libraries that your OpenCL application requires.
  b. Ensure that the file `/opt/Intel/OpenCL/Boards/my_board.fcd` exists in your Custom Platform and contains the name of the vendor-specific libraries (for example, `/data/board_vendor_a/libmy_board_mmd.so`). The FCD loader scans the contents of the `.fcd` file and then loads the `library libmy_board_mmd.so` file from the `board_vendor_a` folder.
- For Intel Arria 10 SoC boards, when you build the SD flash card image for your Custom Platform, create an `Altera.icd` file containing the text `libalteracl.so`. Store the `Altera.icd` file in the `/etc/OpenCL/vendors` directory of your Custom Platform.

  Refer to *Building the Software and SD Card Image for the Intel Arria 10 SoC Custom Platform section of the Intel FPGA SDK for OpenCL Intel Arria 10 SoC Development Kit Reference Platform Porting Guide* for more information.

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

Related Information

Building the Software and SD Card Image for the Intel Arria 10 SoC Custom Platform

6.6.4. Programming an FPGA via the Host

The Intel FPGA SDK for OpenCL Offline Compiler is an offline compiler that compiles kernels independently of the host application. To load the kernels into the OpenCL runtime, include the `clCreateProgramWithBinary` function in your host application.

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

1. Compile your OpenCL kernel with the offline compiler to create the `.aocx` file.
2. Include the `clCreateProgramWithBinary` function in your host application to create the `cl_program` OpenCL program objects from the `.aocx` file.
3. Include the `clBuildProgram` function in your host application to create the program executable for the specified device.

Below is an example host code on using `clCreateProgramWithBinary` to program an FPGA device:

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

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

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

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

4. Create kernel objects from the program executable using the `clCreateKernelsInProgram` or `clCreateKernel` function.
5. Include the kernel execution function to instruct the host runtime to execute the scheduled kernel(s) on the FPGA.
   - To enqueue a command to execute an NDRange kernel, use `clEnqueueNDRangeKernel`.
   - To enqueue a single work-item kernel, use `clEnqueueTask`.

   Attention: Intel recommends that you release an event object when it is not in use. The SDK keeps an event object live until you explicitly instruct it to release the event object. Keeping an unused event object live causes unnecessary memory usage.

   To release an event object, call the `clReleaseEvent` function.

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

For more information on these OpenCL host runtime API calls, refer to the *OpenCL Specification version 1.0*.

Related Information

OpenCL Specification version 1.0

6.6.4.1. 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: Linking your host application to FCD allows you to target multiple FPGA devices from different Custom Platforms. However, this feature has limited support for Custom Platforms that are compatible with SDK versions prior to 16.1.

You can present up to 128 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 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.

Related Information

Accessing Custom Platform-Specific Functions on page 88

6.6.4.1.1. Probing the OpenCL FPGA Devices

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

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

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

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

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

Related Information

Querying the Device Name of Your FPGA Board (diagnose) on page 21

6.6.4.1.2. Querying Device Information

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

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

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

When you query the device information, the host will list your FPGA devices in the following manner: `Device <N>: <board_name>: <name_of_FPGA_board>`

Where:

- *<N>* is the device number.
- *<board_name>* is the board designation you use to target your FPGA device when you invoke the `aoc` command.
- *<name_of_FPGA_board>* is the advertised name of the FPGA board.

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

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

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

Related Information

Listing the Available FPGA Boards in Your Custom Platform (-list-boards) on page 17

6.6.4.1.3. 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);
```

6.6.5. Termination of the Runtime Environment and Error Recovery

In the event that the host application terminates unexpectedly, you must restart the runtime environment and reprogram the FPGA.

The runtime environment is a library that is compiled as part of the host application. When the host application terminates, the runtime environment will also terminate along with any tracking activity that it performs. If you restart the host application, a new runtime environment and its associated tracking activities will reinitialize. The initialization functions reset the kernel's hardware state.

In same cases, unexpected termination of the host application causes the configuration of certain hardware (for example, PCIe hard IP) to be incomplete. To restore the configuration of these hardware, the host needs to reprogram the FPGA.

If you use a Custom Platform that implements customized hardware blocks, be aware that restarting the host application and resetting these blocks might have design implications:

- When the host application calls the `clGetPlatformIDs` function, all kernels and channels will be reset for all available devices.
- When the host application calls the `clGetPlatformIDs` function, it resets FIFO buffers and channels as it resets the device.
- The host application initializes memory buffers via the `clCreateBuffer` and `clEnqueueWriteBuffer` function calls. You cannot access the contents of buffers from a previous host execution within a new host execution.

6.7. Allocating Shared Memory for OpenCL Kernels Targeting SoCs

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

Note:

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

  In DDR memory, shared memory must be physically contiguous. The FPGA cannot consume virtually contiguous memory without a scatter-gather direct memory access (SG-DMA) controller core. The `malloc` function and the `new` operator are for accessing memory that is virtually contiguous.
- CPU caching is disabled for the shared memory.
- When you use shared memory, one copy of the data is used for both the host and the kernel. When this memory is used, OpenCL memory calls are done as zero-copy transfers for buffer reads, buffer writers, maps, and unmaps.

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

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

```
cl_mem src = clCreateBuffer(…, CL_MEM_ALLOC_HOST_PTR, size, …);
int *src_ptr  = (int*)clEnqueueMapBuffer  (…, src, size, …);
*src_ptr = input_value; //host writes to ptr directly
clSetKernelArg (…, src);
clEnqueueNDRangeKernel(…);
clFinish();
```

```
printf (“Result = %d\n”, *dst_ptr); //result is available immediately
clEnqueueUnmapMemObject(…, src, src_ptr, …);
clReleaseMemObject(src); // actually frees physical memory
```

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

Important:

1. If your target board has multiple DDR memory banks, the `clCreateBuffer(..., CL_MEM_READ_WRITE, ...)` function allocates memory to the nonshared DDR memory banks. However, if the FPGA has access to a single DDR bank that is shared memory, then `clCreateBuffer(..., CL_MEM_READ_WRITE, ...)` allocates to shared memory, similar to using the CL_MEM_ALLOC_HOST_PTR flag.
2. The shared memory that you request with the `clCreateBuffer(..., CL_MEM_ALLOC_HOST_PTR, size, ...)` function is allocated in the Linux OpenCL kernel driver, and it relies on the contiguous memory allocator (CMA) feature of the Linux kernel. For detailed information on enabling and configuring the CMA, refer to the *Recompiling the Linux Kernel for the Intel Arria 10 SoC Development Kit* and *Compiling and Installing the OpenCL Linux Kernel Driver* sections of the *Intel FPGA SDK for OpenCL Intel Arria 10 SoC Development Kit Reference Platform Porting Guide* .

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

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

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

- If the host allocates constant memory to shared HPS DDR system and then modifies it after kernel execution, the modifications might not take effect. As a result, subsequent kernel executions might use outdated data. To prevent kernel execution from using outdated constant memory, perform one of the following tasks:
  a. Do not modify constant memory after its initialization.
  b. Create multiple constant memory buffers if you require multiple `__constant` data sets.
  c. If available, allocate constant memory to the FPGA DDR on your accelerator board.

Related Information

- Recompiling the Linux Kernel for the Intel Arria 10 SoC Development Kit
- Compiling and Installing the OpenCL Linux Kernel Driver

6.8. Debugging Your OpenCL System That is Gradually Slowing Down

Your OpenCL system might slow down gradually during execution when a loop in the host application keeps creating events without releasing them. To mitigate this slowdown, the host application must release the `cl_event` objects after they are no longer needed for scheduling or time profiling.

To verify whether the slowdown is caused by the presence of many live events in the OpenCL system, define a context callback function that prints the context callback warnings or errors, as shown in the following example code:

```
void oclContextCallback (const char *errinfo, const void *, size_t, void *) {
     printf ("Context callback: %s\n", errinfo);
}
int main(){
     …
     // Create the context.
     context = clCreateContext (NULL, num_devices, device,
&oclContextCallback, NULL, &status);
     …
}
```

If the number of live events in the system exceeds the threshold limit of 1000 event objects, the callback function prints the following warning message:

```
[Runtime Warning]: Too many 'event' objects in the host. This causes
deterioration in runtime performance.
```

7. Compiling Your OpenCL Kernel

The Intel FPGA SDK for OpenCL offers a list of compiler options that allows you to customize the kernel compilation process. An Intel FPGA SDK for OpenCL Offline Compiler command consists of the `aoc` command, compiler option(s) and settings, and kernel filenames. You can invoke an `aoc` command to direct the compiler to target a specific FPGA board, generate reports, or implement optimization techniques.

Before you compile an OpenCL kernel, verify that the *QUARTUS_ROOTDIR_OVERRIDE* environment variable points to the the Intel Quartus Prime Pro Edition software.

If these environment variables do not have the correct settings, follow the instructions in the *Setting the Intel FPGA SDK for OpenCL Pro Edition User Environment Variables* section of the *Intel FPGA SDK for OpenCL Pro Edition Getting Started Guide* to modify the settings.

Compiling Encrypted Source:

When you compile an encrypted `.cl` file that is provided to you, you can compile only that file with the `aoc` command. You cannot compile multiple encrypted `.cl` files at the same time with the `aoc` command. You cannot compile kernel source files that you have encrypted yourself.

Related Information

- Setting the Intel FPGA SDK for OpenCL Pro Edition User Environment Variables (Windows)
- Setting the Intel FPGA SDK for OpenCL Pro Edition User Environment Variables (Linux)

7.1. Compiling Your Kernel to Create Hardware Configuration File

You can compile an OpenCL kernel and create the hardware configuration file (that is, the `.aocx` file) in a single step.

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

- After you optimize your kernel via the Intel FPGA SDK for OpenCL design flow, and you are now ready to create the `.aocx` file for deployment onto the FPGA.
- You have one or more simple kernels that do not require any optimization.

To compile the kernel and generate the `.aocx` file in one step, invoke the `aoc` *<your_kernel_filename1>*`.cl [`*<your_kernel_filename2>*`.cl ...]` command.

Where [*<your_kernel_filename2>*`.cl` ...] are the optional space-delimited file names of kernels that you can compile in addition to *<your_kernel_filename1>*`.cl`.

The Intel FPGA SDK for OpenCL Offline Compiler groups the `.cl` files into a temporary file. It then compiles this file to generate the `.aocx` file.

Note: If you run the `aoc` command to compile a `.aoco` file (that is, `aoc <your_kernel_filename>.aoco`), the offline compiler generates both a `.aocr` file and a `.aocx` file.

7.2. Compiling Your Kernel without Building Hardware (-c)

To direct the Intel FPGA SDK for OpenCL Offline Compiler to compile your OpenCL kernel and generate the output of the OpenCL parser without creating a hardware configuration file, include the `-c` option in your `aoc` command.

Note: The `-c` flag is not supported in the incremental compilation flow.

- At a command prompt, invoke the `aoc -c <your_kernel_filename1>.cl [<your_kernel_filename2>.cl ...]` command.

  Where [`<your_kernel_filename2>.cl` ...] are the optional space-delimited file names of kernels that you can compile in addition to `<your_kernel_filename1>.cl`.

  When you invoke the `aoc` command with the `-c` flag, the offline compiler compiles the kernel(s) and creates the following files and directories:

  — A `.aoco` file for each `.cl` kernel source file. The offline compiler creates the `.aoco` file(s) in a matter of seconds to minutes.

7.3. Compiling and Linking Your Kernels or Object Files without Building Hardware (-rtl)

To direct the Intel FPGA SDK for OpenCL Offline Compiler to compile your OpenCL kernels (`.cl`), generate a `.aoco` object file for each kernel, and then link them together to create a `.aocr` file without creating a hardware configuration file, include the `-rtl` option in your `aoc` command.

- To compile one or more kernel source files, at a command prompt, invoke the `aoc -rtl <your_kernel_filename1>.cl [<your_kernel_filename2>.cl ...]` command.

  Where [`<your_kernel_filename2>.cl` ...] are the optional space-delimited file names of kernels that you can compile in addition to `<your_kernel_filename1>.cl`.

  When you invoke the `aoc` command with the `-rtl` flag, the offline compiler compiles the kernels and creates the following files and directories:

  — A `.aoco` file for each `.cl` kernel source file. The offline compiler then links them and generates a `.aocr` file. It takes the offline compiler a matter of seconds to minutes to create a `.aoco` file or the `.aocr` file.

  — A `<your_kernel_filename>` folder or subdirectory. It contains intermediate files that the SDK uses to build the hardware configuration file necessary for FPGA programming.

- To compile one or more `.aoco` object files, at a command prompt, invoke the `aoc -rtl <your_kernel_filename>.aoco [<your_kernel_filename2>.aoco ...]` command.

  Where [`<your_kernel_filename2>.aoco` ...] are the optional space-delimited file names of object files that you can compile in addition to `<your_kernel_filename1>.aoco`.

  When you invoke the `aoc` command with the `-rtl` flag, the offline compiler creates the following files and directories:

  — The offline compiler links all the `.aoco` files and generates a `.aocr` file.

  — A `<your_kernel_filename>` folder or subdirectory. It contains intermediate files that the SDK uses to build the hardware configuration file necessary for FPGA programming.

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

To add a directory to the list of directories that the Intel FPGA SDK for OpenCL Offline Compiler searches for header files during kernel compilation, include the `-I=<directory>` option in your `aoc` command.

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

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

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

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

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

or

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

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

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

or

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

7.5. Specifying the Name of an Intel FPGA SDK for OpenCL Offline Compiler Output File (-o <filename>)

To specify the name of a `.aocr` file or a `.aocx` file, include the `-o <filename>` option in your `aoc` command.

- If you implement the multistep compilation flow, specify the names of the output files in the following manner:
  a. To specify the name of the `.aoco` file that the offline compiler creates during an intermediate compilation step, invoke the `aoc -rtl -o <your_object_filename>.aocr <your kernel_filename>.cl` command.
  b. To specify the name of the `.aocx` file that the offline compiler creates during the final compilation step, invoke the `aoc -o <your_executable_filename>.aocx <your_object_filename>.aocr` command.
- If you implement the one-step compilation flow, specify the name of the `.aocx` file by invoking the `aoc -o <your_executable_filename>.aocx <your_kernel_filename>.cl` command.

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

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

Warning: During intermediate compilation with the `-rtl` flag, if you compile a `kernel1.cl` file for board X (that is, `-board=X`) and a `kernel2.cl` file for board Y (that is, `-board=Y`), the Intel FPGA SDK for OpenCL Offline Compiler will issue an error and exit when you run the `aoc -rtl kernel1.aoco kernel2.aoco` command.

When you compile your kernel by including the `-board=<board_name>` option in the `aoc` command, the Intel FPGA SDK for OpenCL Offline Compiler defines the preprocessor macro `AOCL_BOARD_<board_name>` to be 1, which allows you to compile device-optimized code in your kernel.

1. To obtain the names of the available FPGA boards in your Custom Platform, invoke the `aoc -list-boards` command.

   For example, the offline compiler generates the following output:

   ```
   Board List:
   FPGA_board_1
   ```

   where `FPGA_board_1` is the `<board_name>`.

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

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

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

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

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

1. Invoke the following command to generate the `.aoco` file:

   ```
   aoc -rtl -board=FPGA_board_1 <your_kernel_filename>.cl -o
   <my_object_filename>_FPGA_board_1.aocr
   ```

2. Invoke the following command to generate the `.aocx` file:

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

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

```
aoc -board=board_fpga1 kernel_1.cl

aoc -board=board_fpga2 kernel_2.cl
```

Related Information

Specifying the Name of an Intel FPGA SDK for OpenCL Offline Compiler Output File (-o <filename>) on page 107

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

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

When kernel compilation fails because of a fitting constraint problem, the Intel FPGA SDK for OpenCL Offline Compiler displays the following error message:

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

- To overcome this problem, recompile your kernel by invoking the following command:

  `aoc -high-effort <your_kernel_filename>.cl`

  After you invoke the command, the offline compiler displays the following message:

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

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

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

The Intel FPGA SDK for OpenCL Offline Compiler supports preprocessor macros that allow you to pass macro definitions and compile code on a conditional basis.

- To pass a preprocessor macro definition to the offline compiler, invoke the `aoc -D<macro_name> <kernel_filename>.cl` command.
- To override the existing value of a defined preprocessor macro, invoke the `aoc -D<macro_name>=<value> <kernel_filename>.cl` command.

  Consider the following code snippet for the kernel `sum`:

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

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

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

```
  }
  sum[get_global_id(0)] = accum;
}
```

To override the UNROLL_FACTOR of 1 and set it to 4, invoke the `aoc -DUNROLL_FACTOR=4 sum.cl` command. Invoking this command is equivalent to replacing the line `#define UNROLL_FACTOR 1` with `#define UNROLL_FACTOR 4` in the `sum` kernel source code.

- To use preprocessor macros to control how the offline compiler optimizes your kernel without modifying your kernel source code, invoke the `aoc -o <hardware_filename>.aocx -D<macro_name>=<value> <kernel_filename>.cl`

  Where:

  `-o` is the offline compiler option you use to specify the name of the `.aocx` file that the offline compiler generates.

  <hardware_filename> is the name of the `.aocx` file that the offline compiler generates using the preprocessor macro value you specify.

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

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

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

  Compile the kernel multiple times by typing the following commands:

```
aoc –o myKernel_64.aocx –DWORK_GROUP_SIZE=64 myKernel.cl
aoc –o myKernel_128.aocx –DWORK_GROUP_SIZE=128 myKernel.cl
```

7.9. Generating Compilation Progress Report (-v)

To direct the Intel FPGA SDK for OpenCL Offline Compiler to report on the progress of a compilation, include the `-v` option in your `aoc` command.

- To direct the offline compiler to report on the progress of a full compilation, invoke the `aoc -v <your_kernel_filename>.cl` command.

  The offline compiler generates a compilation progress report similar to the following example:

```
aoc: Environment checks are completed successfully.
You are now compiling the full flow!!
aoc: Selected target board a10gx
aoc: Running OpenCL parser....
aoc: OpenCL parser completed successfully.
aoc: Compiling....
aoc: Linking with IP library ...
aoc: First stage compilation completed successfully.
aoc: Setting up project for CvP revision flow....
aoc: Hardware generation completed successfully.
```

- To direct the offline compiler to report on the progress of an intermediate compilation step that does not build hardware, invoke the `aoc -rtl -v <your_kernel_filename>.cl` command.

  The offline compiler generates a compilation progress report similar to the following example:

```
aoc: Environment checks are completed successfully.
aoc: Selected target board a10gx
aoc: Running OpenCL parser....
aoc: OpenCL parser completed successfully.
aoc: Compiling....
aoc: Linking with IP library ...
aoc: First stage compilation completed successfully.
aoc: To compile this project, run "aoc <your_kernel_filename>.aoco"
```

- To direct the offline compiler to report on the progress of a compilation for emulation, invoke the `aoc -march=emulator -v <your_kernel_filename>.cl` command.

  The offline compiler generates a compilation progress report similar to the following example:

```
aoc: Environment checks are completed successfully.
You are now compiling the full flow!!
aoc: Selected target board a10gx
aoc: Running OpenCL parser....ex
aoc: OpenCL parser completed successfully.
aoc: Compiling for Emulation ....
aoc: Emulator Compilation completed successfully.
Emulator flow is successful.
```

Related Information

- Compiling and Linking Your Kernels or Object Files without Building Hardware (-rtl) on page 105
- Emulating and Debugging Your OpenCL Kernel on page 121

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

By default, the Intel FPGA SDK for OpenCL Offline Compiler estimates hardware resource usage during compilation . The offline compiler factors in the usage of external interfaces such as PCIe, memory controller, and DMA engine in its calculations. During kernel compilation, the offline compiler generates an estimated resource usage summary in the `<your_kernel_filename>.log` file within the `<your_kernel_filename>` directory. To review the estimated resource usage summary on-screen, include the `-report` option in the `aoc` command.

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

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

```
+--------------------------------------------------------------------+
; Estimated Resource Usage Summary                                   ;
+----------------------------------------+---------------------------+
; Resource                               + Usage                     ;
+----------------------------------------+---------------------------+
; Logic utilization                      ;   35%                     ;
; ALUTs                                  ;   22%                     ;
; Dedicated logic registers              ;   15%                     ;
; Memory blocks                          ;   29%                     ;
; DSP blocks                             ;    0%                     ;
+----------------------------------------+---------------------------;
```

Related Information

Compiling and Linking Your Kernels or Object Files without Building Hardware (-rtl) on page 105

7.11. Suppressing Warning Messages from the Intel FPGA SDK for OpenCL Offline Compiler (-W)

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

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

7.12. Converting Warning Messages from the Intel FPGA SDK for OpenCL Offline Compiler into Error Messages (-Werror)

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

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

7.13. Removing Debug Data from Compiler Reports and Source Code from the .aocx File (-g0)

By default, the Intel FPGA SDK for OpenCL Offline Compiler includes source information in compiler reports and embeds the source code into the `.aocx` binary when it compiles the `.cl` or `.aoco` file. Include the `-g0` option in the `aoc` command to remove source information from the compiler reports and to remove source code and customer IP information from the `.aocx` file.

- To remove source information in reports and remove source code and customer IP information from the `.aocx` file, invoke the `aoc -g0 <your_kernel_filename>.cl` command.

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

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

Caution: The `-no-interleaving` option requires a global memory type parameter. If you do not specify a memory type, the offline compiler issues an error message.

- To direct the offline compiler to disable burst-interleaving for the default global memory, invoke the `aoc <your_kernel_filename>.cl -no-interleaving=default` command.

  Your accelerator board might include multiple global memory types. To identify the default global memory type, refer to board vendor's documentation for your Custom Platform.
- For a heterogeneous memory system, to direct the offline compiler to disable burst-interleaving of a specific global memory type, perform the following tasks:
  a. Consult the `board_spec.xml` file of your Custom Platform for the names of the available global memory types (for example, DDR and quad data rate (QDR)).
  b. To disable burst-interleaving for one of the memory types (for example, DDR), invoke the `aoc <your_kernel_filename>.cl -no-interleaving=DDR` command.
  The offline compiler enables manual partitioning for the DDR memory bank, and configures the other memory bank in a burst-interleaved fashion.
  c. To disable burst-interleaving for more than one type of global memory buffers, include a `-no-interleaving=<global_memory_type>` option for each global memory type.

  For example, to disable burst-interleaving for both DDR and QDR, invoke the `aoc <your_kernel_filename>.cl -no-interleaving=DDR -no-interleaving=QDR` command.

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

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

Include the `-const-cache-bytes=<N>` flag in your `aoc` command to direct the Intel FPGA SDK for OpenCL Offline Compiler to configure the constant memory cache size (rounded up to the closest power of 2).

The default constant cache size is 16 kB.

- To configure the constant memory cache size, invoke the `aoc -const-cache-bytes=<N> <your_kernel_filename>.cl` command, where <*N*> is the cache size in bytes.

  For example, to configure a 32 kB cache during compilation of the OpenCL kernel `myKernel.cl`, invoke the `aoc -const-cache-bytes=32768 myKernel.cl` command.

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

7.16. Relaxing the Order of Floating-Point Operations (-fp-relaxed)

Include the `-fp-relaxed` option in your `aoc` command to direct the Intel FPGA SDK for OpenCL Offline Compiler 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 offline compiler to execute a balanced tree hardware implementation, invoke the `aoc -fp-relaxed <your_kernel_filename>.cl` command.

7.17. Reducing Floating-Point Rounding Operations (-fpc)

Include the `-fpc` option in your `aoc` command to direct the Intel FPGA SDK for OpenCL Offline Compiler to remove intermediary floating-point rounding operations and conversions whenever possible, and to carry additional bits to maintain precision.

Implementing this optimization control also changes the rounding mode. It rounds towards zero only at the end of a chain of floating-point arithmetic operations (that is, multiplications, additions, and subtractions).

- To direct the offline compiler to reduce the number of rounding operations, invoke the `aoc -fpc <your_kernel_filename>.cl` command.

7.18. Speeding Up Your OpenCL Compilation (-fast-compile)

To save 40-90% of compilation time and quickly create the `.aocx` file of your kernel, include the `-fast-compile` Intel FPGA SDK for OpenCL Offline Compiler command option in your `aoc` command.

The `-fast-compile` feature achieves significant savings in compilation time by lowering optimization efforts.

At the command prompt, invoke the `aoc -rtl <your_kernel_filename1>.cl -fast-compile` command.

Warning: Enabling the `-fast-compile` feature might cause some performance issues such as:

- Higher resource use
- Lower f_{max} and as a result lower application performance
- Lower power efficiency

Intel recommends that you use the `-fast-compile` option for internal development only.

Attention:

- You can only use the `-fast-compile` compiler option to compile OpenCL designs targeting Intel Arria 10 and newer devices.
- After you finalize a design, compile your OpenCL kernel without the `-fast-compile` option over multiple seeds to obtain the best performance.
- Regardless of whether the `-fast-compile` feature is enabled, the initial compilation of any OpenCL system on a new board and with a new version of Intel FPGA SDK for OpenCL Pro Edition takes an additional 45 to 60 minutes to complete. The additional time is used to cache some parts of the compilation for future compilations (this behavior will *not* affect kernel performance). To create this cache, define the environment variable *$AOCL_TMP_DIR* to a writable directory that you can share. By default, this cache is stored in `/var/tmp/aocl/$USER` on Linux and *%USERPROFILE%*`\AppData\Local\aocl` on Windows. You can share this writable directory by setting it to a shared network location.

  After you create the cache, you will not need to create it again for the current version of the Intel FPGA SDK for OpenCL and the current targeted board.

7.19. Compiling Your Kernel Incrementally (-incremental)

To compile the modifications you make to your OpenCL design incrementally, include the `-incremental` Intel FPGA SDK for OpenCL Offline Compiler command option in your `aoc` command.

If you have a large, mutli-kernel system and you only want to modify a single kernel, the Intel FPGA SDK for OpenCL Offline Compiler can reuse the results from a previous compilation, and only synthesize, place, and route the kernel(s) that you have modified. Leveraging this incremental compilation feature allows you to dramatically reduce compilation time.

Attention: Enable the incremental compilation feature for internal development of your OpenCL design only. For best circuit performance on your final design, run a full compilation.

Example incremental compilation flow:

```
aoc -incremental <your_kernel_filename>.cl
/*****Update kernels in your OpenCL design*****/
aoc -incremental -fast-compile <your_kernel_filename>.cl
```

1. Perform an initial setup compilation in a clean directory, with the incremental mode enabled, by invoking the `aoc -incremental <your_kernel_filename>.cl` command.

   Note: You must enable the `-incremental` flag when performing the setup compilation.

   This setup compilation does not reuse any result from a previous compilation. When performing a setup compilation, do not include the `-fast-compile` offline compiler command option in the `aoc` command because it will increase the probability of encountering errors in future incremental compilations.

   Tip: Intel recommends that you perform a fresh setup compilation whenever compilation time is not a concern because it reduces the probability of compilation failures in future incremental compilations. Performing many consecutive incremental compilations increases the probability of compilation failures. It also decreases the hardware performance and efficiency of the generated `.aocx` file.

2. Modify the kernels in your OpenCL design.

   Your design may contain multiple `.cl` files.

3. Perform an incremental compilation on your design. For optimal compilation speed, also include the `-fast-compile` flag in your `aoc` command:

   `aoc -incremental -fast-compile <your_kernel_filename>.cl`

4. Review the **Incremental compile** section of the `report.html` file to verify the changes that the offline compiler has detected.

   The `report.html` file is in the `<your_kernel_filename>/reports` directory.

7.19.1. The Incremental Compile Report

Compiling your OpenCL design with the `-incremental` Intel FPGA SDK for OpenCL Offline Compiler command option instructs the offline compiler to include an **Incremental compile** section in the `report.html` file in the *<your_kernel_filename>* project directory.

The Incremental compile report provides the following metrics on your OpenCL design:

- The `<%> of design not preserved` metric at the bottom of the report provides a quick summary of the overall changes to your design. It is the best predictor of compilation time.

  Note: The FPGA resources listed in the Incremental compile report are calculated based on the estimated area models that the Intel FPGA SDK for OpenCL Offline Compiler produces. The area numbers represent an estimate of the area usage in a standard (that is, non-incremental) compilation. You can use these numbers to gage the area your design will consume in a standard compilation.

  The FPGA resource information might not fully match the final area in the Intel Quartus Prime Pro Edition software compilation reports.

Figure 14. Incremental compile Report for a Setup Compilation

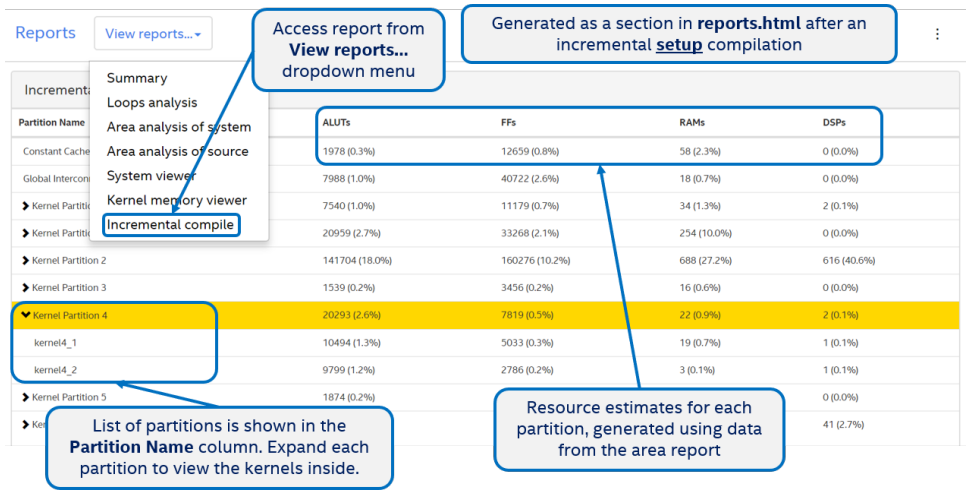
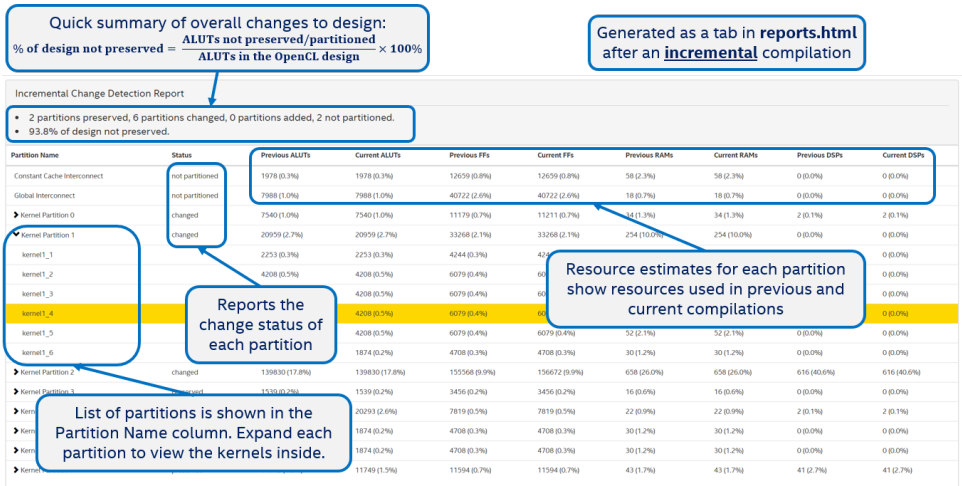


Figure 15. Incremental compile Report for an Incremental Compilation



7.19.2. Additional Command Options for Incremental Compilation

The Intel FPGA SDK for OpenCL incremental compilation feature includes optional functions that you can enable to customize the compilation of your OpenCL design.

Grouping Multiple Kernels into Partitions (-incremental-grouping=<filename>)

By default, the Intel FPGA SDK for OpenCL Offline Compiler places each kernel in your design into a separate partition during incremental compilation. You have the option to group multiple kernels into a single partition by including the `-incremental-grouping=<partition_filename>` command option in your `aoc` command. In general, compilation speed is faster if your design contains fewer partitions.

Example: `aoc -incremental-grouping=<partition_filename> <your_kernel_filename>.cl`

Note: The offline compiler will recompile all kernels in a group even if you modify only one of the kernels. Intel recommends that you group kernels that you will typically modify at the same time.

If your grouped kernels perform many load and store operations, you can speed up compilation further by also including the `-incremental=aggressive` option in your `aoc` command.

The partition file that you pass to the `-incremental-grouping` option is a plain text file. Each line in the file specifies a new partition containing a semi-colon (;)-delimited list of kernel names. For example, the following lines in a partition file specify three partitions, each containing four kernels:

```
reader0;reader1;reader2;reader3
accum0;accum1;accum2;accum3
writer0;writer1;writer2;writer3
```

Compiling a Design in Aggressive Mode (-incremental=aggressive)

To increase the speed of an incremental compilation at the expense of area usage and throughput, include the `-incremental=aggressive` command option in your `aoc` command.

This feature is especially effective when the kernels in your design perform load and store operations to many buffers, or when you have grouped multiple kernels together using the `-incremental-grouping` command option.

Example: `aoc -incremental=aggressive -incremental-grouping=<partition_filename> <your_kernel_filename>.cl`

Caution:

- Enabling the aggressive mode might result in throughput degradations that are larger than what the Fmax degradation indicates.
- For each OpenCL design, avoid changing the compilation mode between incremental compilations. If you compile your design in aggressive mode, enable aggressive mode for all subsequent incremental compilations that you perform on this design. Each time you switch the incremental compilation mode, compilation takes longer to complete.

Specifying a Custom Input Directory (-incremental-input-dir=<path_to_directory>)

During incremental compilation, the offline compiler creates a default <*your_kernel_filename*> project directory in the current working directory to store intermediate compilation files. To base your incremental compilation on a nondefault project directory, specify the directory by including the `-incremental-input-dir=`<*path_to_directory*> command option in your `aoc` command.

You must include the `-incremental-input-dir` option if you compile your design in one or both of the following scenarios:

- Run the `aoc` command from a different working directory than the previous compilation.
- Included the `-o` <*filename*> command option in the previous compilation.

Consider the following example where there is a `mykernel.cl` file in the initial working directory and another revision of the same `mykernel.cl` file in the `new_rev` subdirectory:

```
aoc -incremental mykernel.cl
cd new_rev
aoc -incremental -fast-compile mykernel.cl -incremental-input-dir=../mykernel
```

In this scenario, the offline compiler will reuse the files in `mykernel` project directory from the first compilation as the basis for the second compilation. The offline compiler will create a `new_rev/mykernel` project directory for the second compilation without modifying any file in the original `mykernel` directory.

The `-incremental-input-dir` command option is useful if multiple developers share the same incremental setup compilation. Each developer can run subsequent incremental compilations in their own workspace without overwriting other developers' compilation results.

Disabling Automatic Retry (-incremental-flow=no-retry)

By default, the offline compiler automatically retries a failed incremental compilation by performing a second compilation without preserving any partitions. This second compilation takes longer to complete because it recompiles the entire design.

To disable the offline compiler's automatic retry mechanism, include the `-incremental-flow=no-retry` command option in your `aoc` command. If you enable this feature, the offline compiler will not perform another incremental compilation after the first attempt fails. In addition, the offline compiler will not generate a `.aocx` file.

Enabling this feature allows you to implement your own failure mitigation strategies such as:

- Compiling multiple seeds in parallel to increase the probability of at least one compilation succeeding without retrying.
- Executing a **non-incremental** fast compilation instead of an incremental fast compilation (that is, `aoc -fast-compile` <*your_kernel_filename*>`.cl`).

7.19.3. Limitations of the Incremental Compilation Feature

The Intel FPGA SDK for OpenCL incremental compilation feature is only available to OpenCL designs targeting the Intel Arria 10 FPGAs.

In addition to device support, the incremental compilation has the following limitations:

- You will experience area, Fmax, and power degradations when you enable the incremental compilation feature (`-incremental`) or the fast compilation feature (`-fast-compile`), or both.
- In congested designs, incremental compilations can experience severe (that is, 25% or more) Fmax reductions compared to the initial setup compilation. If the Fmax reduction is unacceptable, perform a non-incremental fast compilation instead to reduce the amount of Fmax degradation while preserving some of the savings in compilation time.
- The offline compiler does not detect changes in RTL libraries that you have included by invoking the `-l <library_name>.aoclib` offline compiler command option. After you modify an RTL library, you must perform a setup compilation again.

  The offline compiler will print a warning message as a reminder to rerun the setup compilation.

7.20. Compiling Your Kernel with Memory Error Correction Coding (-ecc)

Attention: Error correction coding (ECC) is an early Intel FPGA SDK for OpenCL feature that is at the preview stage. Full use of this feature, including the reporting of corrected errors and detected but uncorrected errors, requires an ECC-ready Custom Platform from your board vendor.

Include the `-ecc` option in your `aoc` command to direct the Intel FPGA SDK for OpenCL Offline Compiler to enable error correction coding on the kernel memories (that is, M20ks and MLABs).

The ECC implementation has single error correction and double error detection capabilities for each 32-bit word.

Caution: Enabling the ECC feature will cost an area overhead both in the number of RAMs and ALMs, as well as causing degradation in the Fmax of the system.

- To direct the offline compiler to enable error correction coding hardware implementation, invoke the `aoc -ecc <your_kernel_filename>.cl` command.

8. Emulating and Debugging Your OpenCL Kernel

The Intel FPGA SDK for OpenCL Emulator assesses the functionality of your kernel.

The Intel FPGA SDK for OpenCL Emulator generates a `.aocx` file that executes on x86-64 Windows or Linux host. This feature allows you to emulate the functionality of your kernel and iterate on your design without executing it on the actual FPGA each time. For Linux platform, you can also use the Emulator to perform functional debug.

Caution: Emulation does not support cross-compilation to ARM processor. To run emulation on a design that targets an SoC, emulate on a non-SoC board (for example, `INTELFPGAOCLSDKROOT/board/s5_ref`). When you are satisfied with the emulation results, you may target your design on an SoC board for subsequent optimization steps.

8.1. Modifying Channels Kernel Code for Emulation

The Emulator emulates kernel-to-kernel channels. It does not support the emulation of I/O channels that interface with input or output features of your FPGA board. To emulate applications with a channel that reads or writes to an I/O channel, modify your kernel to add a read or write channel that replaces the I/O channel, and make the source code that uses it is conditional.

The Intel FPGA SDK for OpenCL does not set the `EMULATOR` macro definition. You must set it manually either from the command line or in the source code.

Consider the following kernel example:

```
channel unlong4 inchannel __attribute__((io("eth0_in")));

__kernel void send (int size) {
    for (unsigned i = 0; i < size; i++) {
        ulong4 data = read_channel_intel(inchannel);
        //statements
    }
}
```

To enable the Emulator to emulate a kernel with a channel that interfaces with an I/O channel, perform the following tasks:

1. Modify the kernel code in one of the following manner:
   - Add a matching `write_channel_intel` call such as the one shown below.

```
#ifdef EMULATOR

__kernel void io_in (__global char * restrict arr, int size) {
    for (unsigned i = 0; i < size; i++) {
        ulong4 data = arr[i]; //arr[i] being an alternate data source
        write_channel_intel(inchannel, data);
    }
}
#endif
```

   - Replace the I/O channel access with a memory access, as shown below:

```
__kernel void send (int size) {
    for (unsigned i = 0; i < size; i++) {
        #ifndef EMULATOR

            ulong4 data = read_channel_intel(inchannel);

        #else
            ulong4 data = arr[i]; //arr[i] being an alternate data
source

        #endif
        //statements
    }
}
```

2. Modify the host application to create and start this conditional kernel during emulation.

Related Information

Implementing I/O Channels Using the io Channels Attribute on page 41

8.1.1. Emulating a Kernel that Passes Pipes or Channels by Value

The Intel FPGA SDK for OpenCL Emulator supports a kernel that passes pipes or channels by value.

You may emulate a kernel that passes a channel or pipe by value, as shown in the following example:

```
channel uint my_ch;

void my_function (channel uint ch,
                  __global uint * dst, int i)
{
    dst[i] = read_channel_intel(ch);
}

__kernel void
consumer (__global uint * restrict dst)
{
    for (int i=0;i<5;i++)
    {
        my_function(my_ch, dst, i );
    }
}
```

8.1.2. Emulating Channel Depth

When you compile your OpenCL kernel for emulation, the default channel depth is different from the default channel depth generated when your kernel is compiled for hardware. You can change this behavior when you compile your kernel for emulation with the `-emulator-channel-depth-model` option.

The `-emulator-channel-depth-model` compiler option can take the following values:

default	Channels with an explicit depth attribute have their specified depth. Channels without a specified depth are given a default channel depth that is chosen to provide the fastest execution time for your kernel emulation.
strict	All channel depths in the emulation are given a depth that matches the depth given for the FPGA compilation.
ignore-depth	All channels are given a channel depth chosen to provide the fastest execution time for your kernel emulation. Any explicitly set channel depth attribute is ignored.

8.2. Compiling a Kernel for Emulation (-march=emulator)

To compile an OpenCL kernel for emulation, include the `-march=emulator` option in your `aoc` command.

- Before you perform kernel emulation, perform the following tasks:
  - Install a Custom Platform from your board vendor for your FPGA accelerator boards.
  - Verify that the environment variable *QUARTUS_ROOTDIR_OVERRIDE* points to Intel Quartus Prime Pro Edition software installation folder.
- To emulate your kernels on Windows systems, you need the Microsoft linker and additional compilation time libraries. Verify that the *PATH* environment variable setting includes all the paths described in the *Setting the Intel FPGA SDK for OpenCL Pro Edition User Environment Variables* section of the *Intel FPGA SDK for OpenCL Pro Edition Getting Started Guide*.

  The *PATH* environment variable setting must include the path to the `LINK.EXE` file in Microsoft Visual Studio.
- Ensure that your *LIB* environment variable setting includes the path to the Microsoft compilation time libraries.

  The compilation time libraries are available with Microsoft Visual Studio.
- Verify that the *LD_LIBRARY_PATH* environment variable setting includes all the paths described in the *Setting the Intel FPGA SDK for OpenCL Pro Edition User Environment Variables* section in the *Intel FPGA SDK for OpenCL Pro Edition Getting Started Guide*.

- To create kernel programs that are executable on x86-64 host systems, invoke the `aoc -march=emulator <your_kernel_filename>.cl` command.
- To compile a kernel for emulation that targets a specific board, invoke the `aoc -march=emulator -board=<board_name> <your_kernel_filename>.cl` command.
- For Linux systems, the Intel FPGA SDK for OpenCL Offline Compiler offers symbolic debug support for the debugger.

  The offline compiler's debug support allows you to pinpoint the origins of functional errors in your kernel source code.

Related Information

- Compiling a Kernel for a Specific FPGA Board (-board=<board_name>) on page 107
- Setting the Intel FPGA SDK for OpenCL Pro Edition User Environment Variables (Windows)
- Setting the Intel FPGA SDK for OpenCL Pro Edition User Environment Variables (Linux)

8.3. Emulating Your OpenCL Kernel

To emulate your OpenCL kernel, run the emulation `.aocx` file on the platform on which you build your kernel.

To emulate your kernel, perform the following steps:

1. Run the utility command `aocl linkflags` to find out which libraries are necessary for building a host application. The software lists the libraries for both emulation and regular kernel compilation flows.
2. Build a host application and link it to the libraries from Step 1.

   Attention: To emulate multiple devices alongside other OpenCL SDKs, link your host application to the Khronos ICD Loader Library *before* linking it to the host runtime libraries. Link the host application to the ICD Loader Library by modifying the `Makefile` for the host application. Refer to *Linking Your Host Application to the Khronos ICD Loader Library* for more information.
3. If necessary, move the `<your_kernel_filename>.aocx` file to a location where the host can find easily, preferably the current working directory.
4. To run the host application for emulation:
   - For Windows, first define the number of emulated devices by invoking the `set CL_CONTEXT_EMULATOR_DEVICE_INTELFPGA=<number_of_devices>` command and then run the host application.

     After you run the host application, invoke `set CL_CONTEXT_EMULATOR_DEVICE_INTELFPGA=` to unset the variable.
   - For Linux, invoke the `env CL_CONTEXT_EMULATOR_DEVICE_INTELFPGA=<number_of_devices> <host_application_filename>` command.

This command specifies the number of identical emulation devices that the Emulator needs to provide.

Remember: When the environment variable *CL_CONTEXT_EMULATOR_DEVICE_INTELFPGA* is set, only the emulated devices are available, i.e., access to all physical boards is disabled.

5. If you change your host or kernel program and you want to test it, only recompile the modified host or kernel program and then rerun emulation.

Each invocation of the emulated kernel creates a shared library copy called `<process_ID>-libkernel.so` in a default temporary directory, where `<process_ID>` is a unique numerical value assigned to each emulation run. You may override the default directory by setting the *TMP* or *TEMP* environment variable on Windows, or setting *TMPDIR* on Linux.

Related Information

- Displaying Information on OpenCL Host Runtime and MMD Libraries (link-config or linkflags) on page 93
- Linking Your Host Application to the Khronos ICD Loader Library on page 94

8.4. Debugging Your OpenCL Kernel on Linux

For Linux systems, you can direct the Intel FPGA 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 the host and the kernel seamlessly. You can step through your code, set breakpoints, and examine and set variables.

Prior to debugging your kernel, you must perform the following tasks:

1. During program execution, the debugger cannot step from the host code to the kernel code. You must set a breakpoint before the actual kernel invocation by adding these lines:
   a. `break <your_kernel>`

      This line sets a breakpoint before the kernel.

   b. `continue`

      If you have not begun debugging your host, then type `start` instead.
2. The kernel is loaded as a shared library immediately before the host loads the kernels. The debugger does not recognize the kernel names until the host actually loads the kernel functions. As a result, the debugger will generate the following warning for the breakpoint you set before the execution of the first kernel:

   ```
   Function "<your_kernel>" not defined.
   Make breakpoint pending on future shared library load? (y or [n])
   ```

   Answer `y`. After initial program execution, the debugger will recognize the function and variable names, and line number references for the duration of the session.

Caution: The Emulator uses the OpenCL runtime to report some error details. For emulation, the runtime uses a default print out callback when you initialize a context via the `clCreateContext` function.

Note: Kernel debugging is independent of host debugging. Debug your host code in existing tools such as Microsoft Visual Studio Debugger for Windows and GDB for Linux.

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

1. To generate a `.aocx` file for debugging that targets a specific accelerator board, invoke the `aoc -march=emulator <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 -v <your_kernel_filename>.cl` command. The Intel FPGA SDK for OpenCL Offline Compiler will display the name of the target FPGA board.
2. Run the utility command `aocl linkflags` to find out the additional libraries necessary to build a host application that supports kernel debugging.
3. Build a host application and link it to the libraries from Step 2.
4. Ensure that the `<your_kernel_filename>.aocx` file is in a location where the host can find it, preferably the current working directory.
5. To run the application, invoke the command `env CL_CONTEXT_EMULATOR_DEVICE_INTELFPGA=<number_of_devices> gdb --args <your_host_program_name>`, where `<number_of_devices>` is the number of identical emulation devices that the Emulator needs to provide.
6. If you change your host or kernel program and you want to test it, only recompile the modified host or kernel program and then rerun the debugger.

Related Information

- Compiling a Kernel for a Specific FPGA Board (-board=<board_name>) on page 107
- Generating Compilation Progress Report (-v) on page 110
- Displaying Information on OpenCL Host Runtime and MMD Libraries (link-config or linkflags) on page 93

8.5. Limitations of the Intel FPGA SDK for OpenCL Emulator

The Intel FPGA SDK for OpenCL Emulator feature has some limitations.

1. Execution model

   The Emulator supports the same compilation modes as the FPGA variant. As a result, you must call the `clCreateProgramBinary` function to create `cl_program` objects for emulation.
2. Concurrent execution

Modeling of concurrent kernel executions has limitations. During execution, the Emulator does not actually run interacting work-items in parallel. Therefore, some concurrent execution behaviors, such as different kernels accessing global memory without a barrier for synchronization, might generate inconsistent emulation results between executions.

3. Kernel performance

   The `.aocx` file that you generate for emulation does not include any optimizations. Therefore, it might execute at a significantly slower speed than what an optimized kernel might achieve. In addition, because the Emulator does not implement actual parallel execution, the execution time multiplies with the number of work-items that the kernel executes.

4. The Emulator executes the host runtime and the kernels in the same address space. Certain pointer or array usages in your host application might cause the kernel program to fail, and vice versa. Example usages include indexing external allocated memory and writing to random pointers. You may use memory leak detection tools such as Valgrind to analyze your program. However, the host might encounter a fatal error caused by out-of-bounds write operations in your kernel, and vice versa.

5. Emulation of channel behavior has limitations, especially for conditional channel operations where the kernel does not call the channel operation in every loop iteration. In these cases, the Emulator might execute channel operations in a different order than on the hardware.

8.6. Discrepancies in Hardware and Emulator Results

When you emulate a kernel, your OpenCL system might produce results different from that of the kernel compiled for hardware. You can further debug your kernel before you compile for hardware by running your kernel through simulation.

Warning: These discrepancies usually occur when the Intel FPGA SDK for OpenCL Emulator is unable to model some aspects of the hardware computation accurately, or when your program relies on an undefined behavior.

The most common reasons for differences in emulator and hardware results are as follows:

- Your OpenCL kernel code is using the `#pragma ivdep` directive. The Emulator will not model your OpenCL system when a true dependence is broken by a `pragma ivdep` directive. During a full hardware compilation, you will observe this as an incorrect result.
- Your OpenCL kernel code is relying on uninitialized data. Examples of uninitialized data include uninitialized variables and uninitialized or partially initialized global buffers, local arrays, and private arrays.
- Your OpenCL kernel code behavior depends on the precise results of floating point operations. The Emulator uses floating point computation hardware of the CPU whereas the hardware run uses floating point cores implemented as FPGA cores. The use of `-fp-relaxed` aoc option in your OpenCL kernel code might change the order of operations leading to further divergence in the floating point results.

  Note: The OpenCL standard allows one or more least significant bits of floating point computations to differ between platforms, while still being considered *correct* on both such platforms.
- Your OpenCL kernel code behavior depends on the order of channel accesses in different kernels. The emulation of channel behavior has limitations, especially for conditional channel operations where the kernel does not call the channel operation in every loop iteration. In such cases, the Emulator might execute channel operations in an order different from that on the hardware.
- Your OpenCL kernel or host code is accessing global memory buffers out-of-bounds.

  Attention: — Uninitialized memory read and write behaviors are platform-dependent. Verify sizes of your global memory buffers when using all addresses within kernels, allocating `clCreateBuffer` function call, and transferring `clEnqueueReadBuffer` and `clEnqueueWriteBuffer` function calls.

  — You may use software memory leak detection tools, such as Valgrind, on emulated version of your OpenCL system to analyze memory related problems. Absence of warnings from such tools does not mean the absence of problems. It only means that the tool could not detect any problem. In such a scenario, Intel recommends manual verification of your OpenCL kernel or host code.
- Your OpenCL kernel code is accessing local or private variables out-of-bounds. For example, accessing a local or private array out-of-bounds or accessing a private variable after it has gone out of scope.

  Attention: In software terms, these issues are referred to as stack corruption issues because accessing variables out-of-bounds usually affects unrelated variables located close to the variable being accessed on a software stack. Emulated OpenCL kernels are implemented as regular CPU functions, and have an actual stack that can be corrupted. When targeting hardware, no stack exists and hence, the stack corruption issues are guaranteed to manifest differently. You may use memory leak analyzer tools, such as Valgrind, when a stack corruption is suspected. However, stack related issues are usually difficult to identify. Intel recommends manual verification of your OpenCL kernel code to debug a stack related issue.

- Your OpenCL kernel code is using shifts that are larger than the type being shifted. For example, shifting a 64-bit integer by 65 bits. According to the OpenCL specification version 1.0, the behavior of such shifts is undefined.

  Warning: If the shift amount is known during compilation, the offline compiler issues a warning message. You must heed to the warning message.
- When you compile your OpenCL kernel for emulation, the default channel depth is different from the default channel depth generated when your kernel is compiled for hardware. This difference in channel depths might lead to scenarios where execution on the hardware hangs while kernel emulation works without any issue. Refer to Emulating Channel Depth on page 123 for information on how to fix the channel depth difference.
- In terms of ordering the printed lines, the output of the `printf` function might be ordered differently on the Emulator and hardware. This is because, in the hardware, `printf` data is stored in a global memory buffer and flushed from the buffer only when the kernel execution is complete, or when the buffer is full. In the Emulator, the `printf` function uses the x86 `stdout`.
- If you perform an unaligned load/store through upcasting of types, the FPGA and emulator might produce different results. An load/store of this type is undefined in the C99 specification.

  For example, the following operation might produce unexpected results:

```
int tmp = *((int *) (my_ptr + 5));
```

Related Information

Debugging Your OpenCL Library Through Simulation (Preview) on page 161

8.7. Using the Fast Emulator (Preview)

A preview version of a fast OpenCL emulator is available with the Pro edition of the Intel FPGA SDK for OpenCL. This feature enables faster emulation of OpenCL kernels, and supports most of the same features as the default emulator. Review this section to see whether your kernel code is compatible with this preview release of the fast emulator.

Fast Emulator Requirements

The fast emulator supports 64-bit Windows and Linux operating systems. On Linux systems, the GNU C Library (glibc) version 2.15 or greater is required. Some older Linux systems might not meet this requirement.

Fast Emulator Setup

If you installed the Intel FPGA SDK for OpenCL Pro edition with administrator privileges, no additional setup is needed.

If you did not install the Intel FPGA SDK for OpenCL with administrator privileges, you must perform some additional steps to enable the fast emulator:

1. Manually set up the fast emulator installable client driver (ICD) entry:

- **Linux**: Ensure that the file `/etc/OpenCL/vendors/Intel_FPGA_SSG_Emulator.icd` matches the file found in the directory that the environment variable INTELFPGAOCLSDKROOT specifies The INTELFPGAOCLSDKROOT environment variable points to the location of the SDK installation.

  If the files do not match, or if it is missing from `/etc/OpenCL/vendors`, copy the `Intel_FPGA_SSG_Emulator.icd` file from the location specified by the INTELFPGAOCLSDKROOT environment variable to `/etc/OpenCL/vendors` directory.
- **Windows**: Ensure that the registry key `HKEY_LOCAL_MACHINE\SOFTWARE\Khronos\OpenCL\Vendors` contains the following value:
  ```
  [HKEY_LOCAL_MACHINE\SOFTWARE\Khronos\OpenCL\Vendors]
  "intelocl64_emu.dll"=dword:00000000
  ```

2. Manually install (or reinstall) the Intel FPGA SDK for OpenCL with Intel FPGA Support Preview.

   This step ensures that the 64-bit Intel SDK for OpenCL - Offline Compiler command line interface (`ioc64`) is installed on your system.

Using the Fast Emulator

When you target the fast emulator, the commands for compiling your kernel and host code are different compared to the default emulator.

Kernel Compilation

To compile a kernel to use with the fast emulator, specify the `--fast-emulator` flag to the compiler invocation command. For example:

```
aoc -march=emulator -fast-emulator <kernel_filename>.cl -o
<kernel_filename>.aocx
```

This command generates an `.aocx` file that can be used with the fast emulator. This `.aocx` file is not compatible with the default emulator.

Host Compilation

To use the fast emulator, your host code must select the fast emulator OpenCL platform. Your code can select this platform by using the following fast emulator platform name:

```
Intel(R) FPGA Emulation Platform for OpenCL(TM) (preview)
```

When you compile the host code, ensure you link against the Khronos ICD Loader Library, as described in Linking Your Host Application to the Khronos ICD Loader Library on page 94.

See the OpenCL Vector Addition Design Example on the Intel FPGA website for an example application that makes use of the fast emulator.

Related Information

- Linking Your Host Application to the Khronos ICD Loader Library on page 94
- OpenCL Installable Client Driver (ICD) Loader
- OpenCL Vector Addition Design Example

8.7.1. Fast Emulator Environment Variables

Several environment variables are available to modify the behavior of the fast emulator.

OCL_TBB_NUM_WORKERS

Indicates a maximum number of threads that can be used by the emulator. The default value is 32, and the maximum value is 255. Each thread can run a single kernel.

If the application requires several kernels to be executing simultaneously, the OCL_TBB_NUM_WORKERS should be set appropriately (to the number of kernels used or a higher value).

CL_CONFIG_CPU_FORCE_LOCAL_MEM_SIZE

Set the amount of available OpenCL local memory, with units, for example: `8MB`, `256KB`, or `1024B`.

CL_CONFIG_CPU_FORCE_PRIVATE_MEM_SIZE

Set the amount of available OpenCL private memory, with units, for example: `8MB`, `256KB`, or `1024B`.

CL_CONFIG_CHANNEL_DEPTH_EMULATION_MODE

When you compile your OpenCL kernel for emulation, the channel depth is different from the channel depth generated when your kernel is compiled for hardware. You can change this behavior with the CL_CONFIG_CHANNEL_DEPTH_EMULATION_MODE environment variable.

This CL_CONFIG_CHANNEL_DEPTH_EMULATION_MODE environment variable can take the following values:

`ignore-depth`	All channels are given a channel depth chosen to provide the fastest execution time for your kernel emulation. Any explicitly set channel depth attribute is ignored. This value is used by default if CL_CONFIG_CHANNEL_DEPTH_EMULATION_MODE environment variable is not set.
`default`	Channels with an explicit depth attribute have their specified depth. Channels without a specified depth are given a default channel depth that is chosen to provide the fastest execution time for your kernel emulation.
`strict`	All channel depths in the emulation are given a depth that matches the depth given for the FPGA compilation.

For channels, the CL_CONFIG_CHANNEL_DEPTH_EMULATION_MODE environment variable should be set for a kernel compilation. For pipes, it should be set for a host program execution.

Related Information

Emulating Channel Depth on page 123

8.7.2. Extensions Supported by the Fast Emulator

The fast emulator offers varying levels of support for different OpenCL extensions.

The following OpenCL extensions are fully supported by the fast emulator:

- `cl_intel_fpga_host_pipe`
- `cl_khr_3d_image_writes`
- `cl_khr_byte_addressable_store`
- `cl_khr_icd`
- `cles_khr_int64`

The fast emulator also supports the following OpenCL extensions to a similar degree as the default emulator:

- `cl_intel_channels`
- `cl_khr_local_int32_base_atomics`
- `cl_khr_local_int32_extended_atomics`
- `cl_khr_global_int32_base_atomics`
- `cl_khr_global_int32_extended_atomics`
- `cl_khr_fp64`
- `cl_khr_3d_image_writes`
- `cl_khr_fp16`

8.7.3. Fast Emulator Known Issues

A few known issues might affect your use of the fast emulator. Review these issues to avoid possible problems when using the fast emulator.

AutoRun Kernels

AutoRun kernels shut down only after a host program exits, not after a `clReleaseProgram()` call.

Compiler Diagnostics

Some compiler diagnostics are not yet implemented for the fast emulator.

CL_OUT_OF_RESOURCES Error Returned From `clEnqueueNDRangeKernel()`

This can occur when the kernel used more `__private` or `__local` memory than the fast emulator supports by default.

Try setting the CL_CONFIG_CPU_FORCE_PRIVATE_MEM_SIZE or the CL_CONFIG_CPU_FORCE_LOCAL_MEM_SIZE environment variables, as described in Fast Emulator Environment Variables on page 131.

CL_INVALID_VALUE Error Returned From `clCreateKernel()`

It is possible a call to the `clBuildProgram()` was missed.

This call is required by the OpenCL specification, even if a program is created from a binary. See section 5.4.2 of the OpenCL Specification version 1.0 for details.

Related Information

- Fast Emulator Environment Variables on page 131
- The OpenCL Specification, Version 1.0

9. Reviewing Your Kernel's report.html File

Attention: The `analyze-area` Intel FPGA SDK for OpenCL utility option has been deprecated. To view your kernel's estimated area usage, refer to the `report.html` file.

For reference information on the deprecated area report, refer to the *Review Your Kernel's Area Report to Identify Inefficiencies in Resource Usage* section in version 16.0 of the *Altera SDK for OpenCL Best Practices Guide*.

After compiling your OpenCL kernel, the Intel FPGA SDK for OpenCL Offline Compiler automatically generates an HTML report that analyzes various aspects of your kernel, such as area, loop structure, memory usage, and kernel pipeline.
To launch the HTML report, open the `report.html` file in the `<your_kernel_filename>/reports` directory.

For more information on the HTML report, refer to the *Review Your Kernel's report.html File* section in the *Intel FPGA SDK for OpenCL Best Practices Guide*.

Related Information

- Review Your Kernel's report.html File
- Altera SDK for OpenCL Best Practices Guide version 16.0

10. Profiling Your OpenCL Kernel

The Intel FPGA Dynamic Profiler for OpenCL measures and reports performance data collected during OpenCL kernel execution on the FPGA. The Intel FPGA Dynamic Profiler for OpenCL relies on performance counters to gather kernel performance data. You can then review performance data in the Profiler GUI.

10.1. Instrumenting the Kernel Pipeline with Performance Counters (-profile)

To instrument the OpenCL kernel pipeline with performance counters, include the `-profile=(all|autorun|enqueued)` option of the `aoc` command when you compile your kernel.

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

- To instrument the Verilog code in the <*your_kernel_filename*>.aocx file with performance counters, invoke the `aoc -profile=(all|autorun|enqueued) <your_kernel_filename>.cl` command, where:
  - `all` argument instruments all kernels in the <*your_kernel_filename*>.cl file with performance counters. This is the default option if no argument is provided.
  - `autorun` argument instruments only the autorun kernels with performance counters.
  - `enqueued` argument instruments only the non-autorun kernels with performance counters.

  Attention: — When profiling multiple, different kernels, do not use the same kernel names across different `.aocx` files. If the kernel names are the same, the profile data will be wrong for these kernels.

  — Regardless of the input to the `clGetProfileDataDeviceIntelFPGA` host library call , the Intel FPGA Dynamic Profiler for OpenCL only profiles kernel types that you indicate during compilation.

  Caution: Profiling autorun kernels results in some hardware overhead for the counters. For large designs, the overhead can cause f_{max} and design frequency degradation. It can also lead to designs that cannot fit on the chip if the Intel FPGA Dynamic Profiler for OpenCL profiles every kernel.
- Run your host application from a local disk to execute the <*your_kernel_filename*>.aocx file on your FPGA. During kernel execution, the performance counters throughout the kernel pipeline collects profile information. The host saves the information in a `profile.mon` monitor description file in your current working directory.

  Caution: Because of slow network disk accesses, running the host application from a networked directory might introduce delays between kernel executions. These delays might increase the overall execution time of the host application. In addition, they might introduce delays between kernel launches while the runtime stores profile output data to disk.

10.2. Launching the Intel FPGA Dynamic Profiler for OpenCL GUI (report)

You can use the Intel FPGA Dynamic Profiler for OpenCL `report` utility command to launch the Profiler GUI. The Profiler GUI allows you to view kernel performance data statistics that the Intel FPGA Dynamic Profiler for OpenCL collects during kernel execution.

The Intel FPGA Dynamic Profiler for OpenCL stores performance data in a `profile.mon` file in your current working directory.

- To launch the Intel FPGA Dynamic Profiler for OpenCL GUI, invoke the `aocl report <your_kernel_filename>.aocx profile.mon [<your_kernel_filename>.source]` utility command.

  Important: If you do not specify the `.source` file in the command, the Intel FPGA Dynamic Profiler for OpenCL GUI will not have a source code tab.

10.3. Profiling Autorun Kernels

Autorun kernel profiling feature allows you to profile autorun kernels.

Kernels that are marked with the `autorun` attribute are referred to as autorun kernels. Hence, an autorun kernel starts executing automatically before other kernels are launched explicitly by the host, and restarts automatically on completion. For more information about the `autorun` attribute, refer to *Omit Communication Hardware between the Host and the Kernel* topic.

Since autorun kernels never complete, you must call the host library call `clGetProfileDataDeviceIntelFPGA` to capture the autorun profiler data. You can instruct the host application to make this call at any point during execution.

Attention: Autorun kernel profiling feature does not support autorun kernels that use global memory or allow profiling individual kernels.

Related Information

- Omit Communication Hardware between the Host and the Kernel on page 171
- Profiling Enqueued and Autorun Kernels on page 86
- Profile Data Acquisition on page 87
- Multiple Autorun Profiling Calls on page 87

11. Developing OpenCL Applications Using Intel Code Builder for OpenCL

The Intel Code Builder for OpenCL is a software development tool available as part of the Intel FPGA SDK for OpenCL. It enables development of OpenCL applications via well-known integrated development environments targeting the Intel FPGAs.

The Intel Code Builder for OpenCL provides a set of Microsoft Visual Studio and Eclipse plug-ins that enable capabilities for creating, building, debugging, and analyzing Windows and Linux applications accelerated with OpenCL.

11.1. Configuring the Intel Code Builder for OpenCL Offline Compiler Plug-in for Microsoft Visual Studio

To enable the Intel Code Builder for OpenCL offline compiler plug-in for Microsoft Visual Studio, perform the following steps:

1. In the Visual Studio software, select **Project ➤ Properties**.
2. In the **Project ➤ Properties ➤ Code Builder** page, change the **Device** to your desired FPGA device.
3. In the **C/C++ ➤ General** property page, under **Additional Include Directories**, enter the full path to the directory where the OpenCL code header files are located (`$(`*`INTELFPGAOCLSDKROOT`*`)\include`).
4. In the **Linker ➤ General** property page, under **Additional Library Directories**, enter the full path to the directory where the OpenCL code run-time import library file is located. For example, for 64-bit application, add `$(`*`INTELFPGAOCLSDKROOT`*`)\lib\x64`:
5. In the **Linker ➤ Input** property page, under **Additional Dependencies**, enter the name of the OpenCL ICD import library file as `OpenCL.lib`.

11.2. Configuring the Intel Code Builder for OpenCL Offline Compiler Plug-in for Eclipse

To enable the Intel Code Builder for OpenCL offline compiler plug-in for Eclipse IDE, perform the following steps:

1. Copy the `CodeBuilder_<`*`version`*`>.jar` plug-in file from `$`*`INTELFPGAOCLSDKROOT`*`/eclipse-plug-in` to *<ECLIPSE_ROOT_FOLDER>*`/dropins`.

Attention: In Linux, you must add $*INTELFPGAOCLSDKROOT*\bin to the LD_LIBRARY_PATH environment variable.

2. Run the Eclipse IDE.
3. Select **Windows ➤ Preferences**.
4. Switch to the Intel OpenCL dialog.
5. Set the OpenCL binary directory to $*INTELFPGAOCLSDKROOT*/bin.

   Once the offline compiler is configured, you can use the **Code-Builder** menu to perform the following basic operations:

   - Create a new session
   - Open an existing session
   - Save a session
   - Build a session
   - Compile a session
   - Configure a session

   For more information about the Intel Code Builder for OpenCL, refer to *Developer Guide for Intel SDK for OpenCL Applications*. For information about how to configure the Intel Code Builder for OpenCL for Microsoft Visual Studio, refer to *Intel Code Builder for OpenCL API for Microsoft Visual Studio*. For information about how to configure the Intel Code Builder for OpenCL for Eclipse, refer to *Intel Code Builder for OpenCL API for Eclipse*.

Related Information

- Developer Guide for Intel SDK for OpenCL Applications
- Intel Code Builder for OpenCL API for Microsoft Visual Studio
- Intel Code Builder for OpenCL API for Eclipse

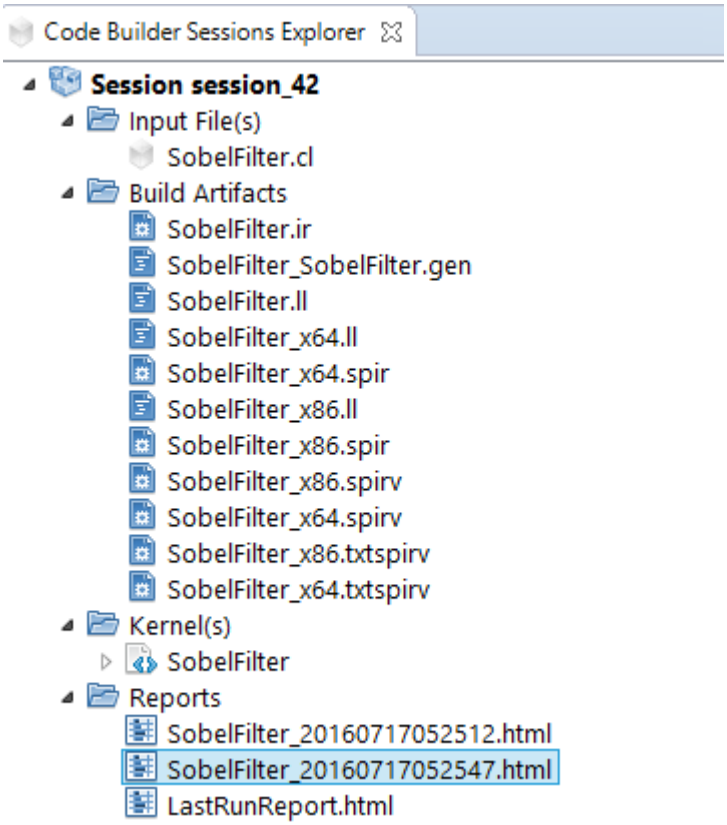
11.3. Creating a Session in the Intel Code Builder for OpenCL

Perform the following steps to create a session in the Intel Code Builder for OpenCL:

1. Select **Code-Builder ➤ OpenCL Kernel Development ➤ New Session**.
2. Specify the session name, path to the folder to store the session file and the content of the session (can be either an empty session or with a predefined OpenCL code).
3. Click **Done**.

Once the session is created, the new session appears in the **Code Builder Sessions Explorer** view.

Figure 16. Code Builder Sessions Explorer



Note: If you do not see the **Code Builder Session Explorer** view, select **Code-builder ➤ OpenCL Kernel Development ➤ Windows ➤ Code Builder Session Explorer**.

11.4. Configuring a Session

A configuration is a set of analysis inputs such as assigned variables, number of iterations, global sizes and local sizes of a specific kernel, and so on. You can create a separate configuration for each set of inputs that you want to analyze.

You can configure a session by right-clicking the session in the **Code Builder Session Explorer** and selecting **Session Options**. Alternatively, you can also open the **Session Settings** dialog box by selecting **Code-Builder ➤ OpenCL Kernel Development ➤ Session Options**.

The **Session Settings** dialog box allows you to configure:

- Device options such as target machine, OpenCL platform, and OpenCL device.
- Build options such as offline compiler flags and build architecture.
- Build artifacts such as `.aocx` and `.aoco` files, and static reports.
- General options such as job architecture and network settings.

In the **Device Options** tab, ensure to select **Intel FPGA SDK for OpenCL** in the OpenCL platform drop-down list.

Under the **Build Options** tab, in the **OpenCL Build Options** section, enter the Intel FPGA SDK for OpenCL Offline Compiler flags manually.

Attention: If your kernel has channels, you must configure workflows. A workflow is a set of kernels, which can be executed sequentially. Workflow can be used to execute a workload with channels where you connect the input of one kernel with the output of the previous kernel (by assigning the same variable for both kernels).

For more information about configuring a session and variable management, refer to the *Developer Guide for Intel SDK for OpenCL Applications*.

Related Information

- Configuring a Session in Microsoft Visual Studio
- Configurations and Settings in Eclipse
- Variable Management in Microsoft Visual Studio
- Variable Management in Eclipse

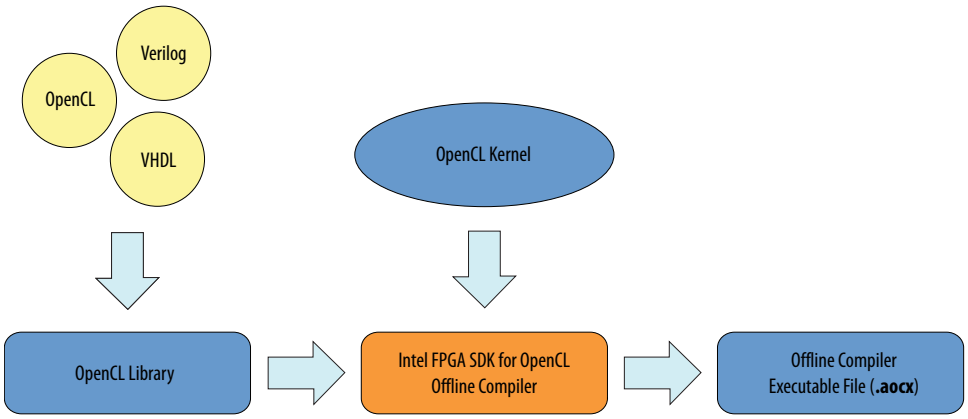
12. Intel FPGA SDK for OpenCL Advanced Features

The Intel FPGA SDK for OpenCL provides advanced features you can use to control the following aspects of the design architecture and the Intel FPGA SDK for OpenCL Offline Compiler's behavior:

12.1. OpenCL Library

An OpenCL library is a single file that contains multiple functions. Each function is comprised of data processing logic that works at any clock frequency. You can create an OpenCL library in OpenCL or register transfer level (RTL). You can then include this library file and use the functions inside your OpenCL kernels.

Figure 17. Overview of Intel FPGA SDK for OpenCL's Library Support



You may use a previously-created library or create your own library. To use an OpenCL library, you do not require in-depth knowledge in hardware design or in the implementation of library components. To create an OpenCL library, you need to create the following files and components:

Table 4. Necessary Files and Components for Creating an OpenCL Library

File or Component	Description
RTL Components	
RTL source files	Verilog, System Verilog, or VHDL files that define the RTL component. Additional files such as Intel Quartus Prime IP File (`.qip`), Synopsys Design Constraints File (`.sdc`), and Tcl Script File (`.tcl`) are not allowed.

continued...

File or Component	Description
eXtensible Markup Language File (`.xml`)	Describes the properties of the RTL component. The Intel FPGA SDK for OpenCL Offline Compiler uses these properties to integrate the RTL component into the OpenCL pipeline.
Header file (`.h`)	A C-style header file that declares the signatures of function(s) that are implement by the RTL component.
OpenCL emulation model file (`.cl`)	Provides C model for the RTL component that is used only for emulation. Full hardware compilations use the RTL source files.
OpenCL Functions	
OpenCL source files (`.cl`)	Contains definitions of the OpenCL functions. These functions are used during emulation and full hardware compilations.
Header file (`.h`)	A C-style header file that declares the signatures of function(s) that are defined in the OpenCL source files.

Remember: There is no difference in the header file used for RTL and OpenCL library functions. A single header file can have both types of functions declared. A single library can contain both RTL and OpenCL library functions.

Understanding RTL Modules and the OpenCL Pipeline on page 143
Packaging an OpenCL Helper Function File for an OpenCL Library on page 157
Packaging an RTL Component for an OpenCL Library on page 157
Verifying the RTL Modules on page 159
Packaging Multiple Object Files into a Library File on page 160
Specifying an OpenCL Library when Compiling an OpenCL Kernel on page 160
Debugging Your OpenCL Library Through Simulation (Preview) on page 161
Using an OpenCL Library that Works with Simple Functions (Example 1) on page 164
Using an OpenCL Library that Works with External Memory (Example 2) on page 165
OpenCL Library Command-Line Options on page 166

Related Information

OpenCL Library Command-Line Options on page 166

12.1.1. Understanding RTL Modules and the OpenCL Pipeline

The OpenCL library feature allows you to use RTL modules, written in Verilog, SystemVerilog, or VHDL, inside OpenCL kernels. This section provides an overview of how the Intel FPGA SDK for OpenCL Offline Compiler integrates RTL modules into the Intel FPGA SDK for OpenCL pipeline architecture.

Use RTL modules under the following circumstances:

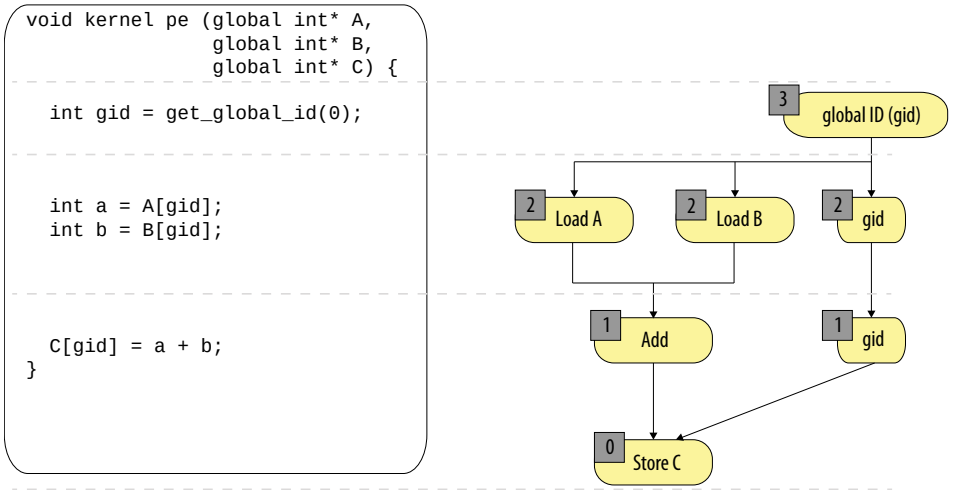
- You want to use optimized and verified RTL modules in OpenCL kernels without rewriting the modules as OpenCL functions.
- You want to implement OpenCL kernel functionality that you cannot express effectively in OpenCL.

12.1.1.1. Overview: Intel FPGA SDK for OpenCL Pipeline Approach

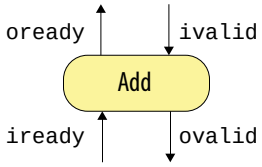
The following figure depicts the architecture of an Intel FPGA SDK for OpenCL pipeline:

Figure 18. Parallel Execution Model of Intel FPGA SDK for OpenCL Pipeline Stages

The operations on the right represent the SDK's pipeline implementation of the OpenCL kernel code on the left. Each yellow box is an operation or data value found in the pipeline. The number associated with each operation represents the number of threads in the pipeline.



Assume each level of operation is one stage in the pipeline. At each stage, the Intel FPGA SDK for OpenCL Offline Compiler executes all operations in parallel by the thread existing at that stage. For example, thread 2 executes Load A, Load B, and copies the current global ID (via `gid`) to the next pipeline stage. Similar to the pipelined execution on instructions in reduced instruction set computing (RISC) processors, the SDK's pipeline stages also execute in parallel. The threads will advance to the next pipeline stage only after all the stages have completed execution.



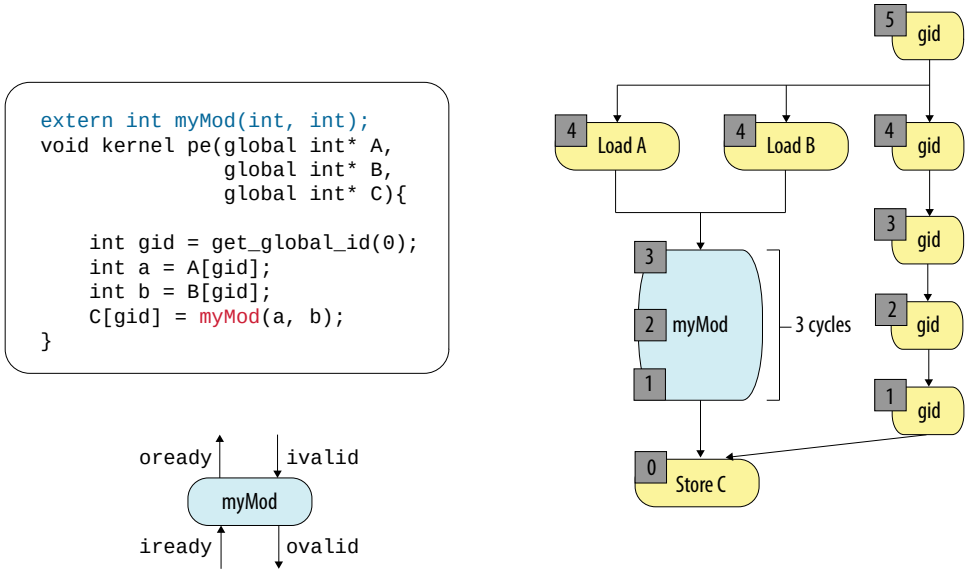
Some operations are capable of stalling the Intel FPGA SDK for OpenCL pipeline. Examples of such operations include variable latency operations like memory load and store operations. To support stalls, ready and valid signals need to propagate throughout the pipeline so that the offline compiler can schedule the pipeline stages. However, ready signals are not necessary if all operations have fixed latency. In these cases, the offline compiler optimizes the pipeline to statically schedule the operations, which significantly reduces the logic necessary for pipeline implementation.

12.1.1.2. Integration of an RTL Module into the Intel FPGA SDK for OpenCL Pipeline

When you specify an OpenCL library during kernel compilation, the offline compiler integrates the RTL module within the library into the overall pipeline.

Figure 19. Integration of an RTL Module into an Intel FPGA SDK for OpenCL Pipeline

This figure depicts the integration of the RTL module `myMod` into the pipeline depicted in Figure 18 on page 144.



The depicted RTL module has a balanced latency where the threads of the RTL module match the number of pipeline stages. A balanced latency allows the threads of the RTL module to execute without stalling the SDK's pipeline.

Setting the latency of the RTL module in the RTL specification file allows the offline compiler to balance the pipeline latency. RTL supports Avalon™ Streaming (Avalon-ST) interfaces; therefore, the latency of the RTL module can be variable (that is, not fixed). However, the variability in the latency should be small in order to maximize performance. In addition, specify the latency in the *<RTL module description file name>*`.xml` specification file so that the RTL module experiences a good approximation of the actual latency in steady state.

Related Information

- Avalon Interface Specifications
- Pipelined Read Transfer with Variable Latency
- Pipelined Read Transfers with Fixed Latency
- Avalon Streaming (Avalon-ST) Interface on page 147
- XML Syntax of an RTL Module on page 149

12.1.1.3. Stall-Free RTL

The Intel FPGA SDK for OpenCL Offline Compiler can optimize hardware resource usage and performance by removing stall logic around an RTL module with fixed latency.

An RTL module that has a variable latency and uses Avalon-ST input and output signals can wait until input data is ready. Conversely, the Intel FPGA SDK for OpenCL pipeline can stall until it receives valid output data from the RTL module. For an RTL module with a fixed latency, you can remove an RTL stall by modifying the <*RTL module description file name*>.xml specification file, as described below.

1. To instruct the offline compiler to remove stall logic around the RTL module, if appropriate, set the IS_STALL_FREE attribute under the FUNCTION element to "yes".

   This modification informs the offline compiler that the RTL module produces valid data every EXPECTED_LATENCY cycle(s).

   Note: EXPECTED_LATENCY is an attribute you specify in the .xml file under the FUNCTION element.

2. Specify a value for EXPECTED_LATENCY such that the latency equals the number of pipeline stages in the module.

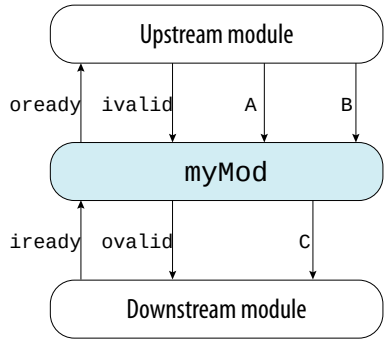
   Caution: An inaccurate EXPECTED_LATENCY value will cause the RTL module to be out of sync with the rest of the pipeline.

A stall-free RTL module might receive an invalid input signal (that is, ivalid is low). In this case, the module ignores the input and produces invalid data on the output. For a stall-free RTL module without an internal state, it might be easier to propagate the invalid input through the module. However, for an RTL module with an internal state, you must handle an ivalid=0 input carefully.

12.1.1.4. RTL Module Interfaces

For an RTL module to properly interact with other compiler-generated operations, you must support a simplified Avalon Streaming Interface at both input and output of an RTL module.

The following diagram shows the complete interface of the myMod RTL module shown in Figure 19 on page 145.



In this diagram, myMod interacts with the upstream module through data signals, A and B, and control signals, ivalid (input) and oready (output). The ivalid control signal equals 1 (ivalid = 1) if and only if data signal A and data signal B contain valid data. When the control signal oready equals 1 (oready = 1), it indicates that the myMod RTL module can process the data signals A and B if they are valid (that is, ivalid = 1). When ivalid = 1 and oready = 0, the upstream module is expected to hold the values of ivalid, A, and B in the next clock cycle.

myMod interacts with the downstream module through data signal C and control signals, ovalid (output) and iready (input). The ovalid control signal equals 1 (ovalid = 1) if and only if data signal C contains valid data. The iready control signal equals 1 (ivalid = 1) indicates that the downstream module is able to process data signal C if it is valid. When ovalid = 1 and iready = 0, the myMod RLT module is expected to hold the valid of the ovalid and C signals in the next clock cycle.

myMod module will assert oready for a single clock cycle to indicate it is ready for an active cycle. Cycles during which myMod module is ready for data are called ready cycles. During ready cycles, the module above myMod module can assert ivalid to send data to myMod.

For a detailed explanation of data transfer under backpressure, refer to "Data Transfer with Backpressure" in *Avalon Interface Specifications*. Ignore the information pertaining to readyLatency option.

Related Information

Data Transfer with Backpressure

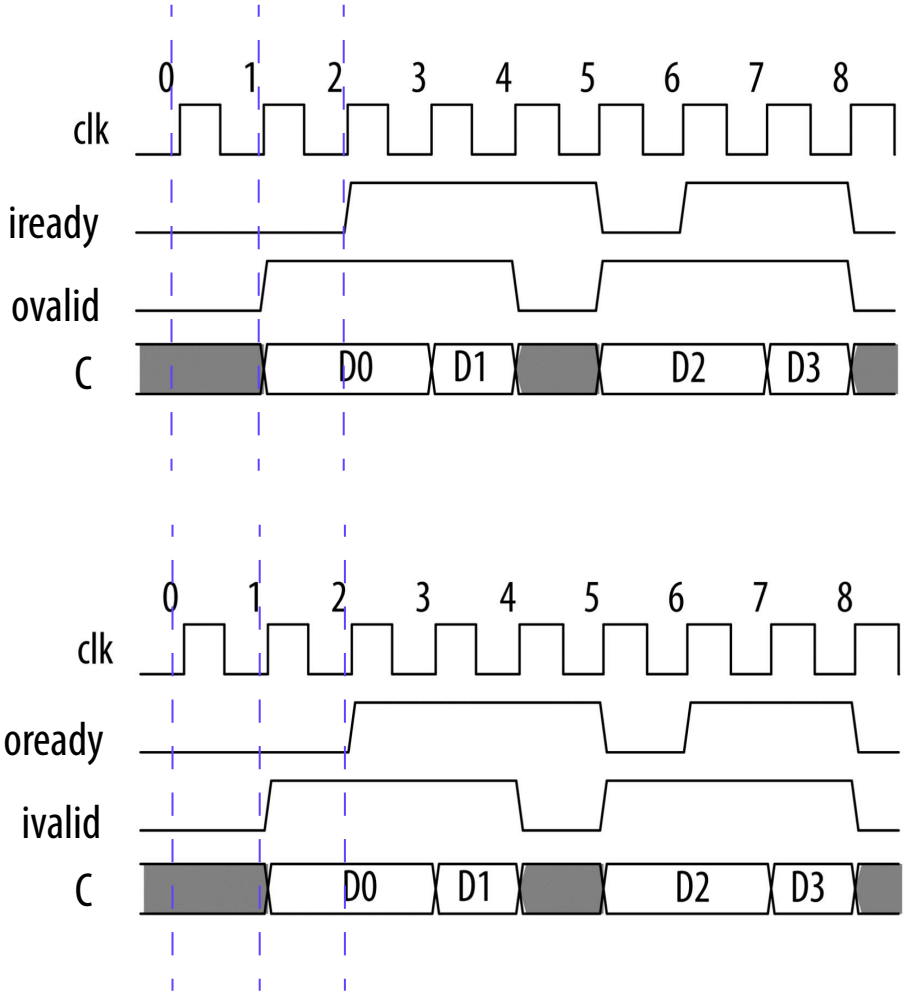
12.1.1.5. Avalon Streaming (Avalon-ST) Interface

The offline compiler expects the RTL module to support Avalon-ST interface with readyLatency = 0, at both input and output.

As shown in Figure 19 on page 145, the RTL module must have 4 ports:

- ivalid and iready, as the input Avalon-ST interface
- ovalid and oready, as the output Avalon-ST interface

The following figure shows the timing diagram for input data transfer with back pressure. For more information about Avalon-ST interfaces, see the "Avalon Streaming Interfaces" section in *Avalon Interface Specifications*.



For an RTL module with a fixed latency, the output signals (`ovalid` and `oready`) can have constant high values, and the input ready signal (`iready`) can be ignored.

A stall-free RTL module might receive an invalid input signal (`ivalid` is low). In this case, the module ignores the input and produces invalid data on the output. For a stall-free RTL module without an internal state, it might be easier to propagate the invalid input through the module. However, for an RTL module with an internal state, you must handle an `ivalid = 0` input carefully.

12.1.1.6. RTL Reset and Clock Signals

Resets and clocks of RTL modules are connected to the same clock and reset drivers as the rest of the OpenCL pipeline.

Because of the common clock and reset drivers, an RTL module runs in the same clock domain as the OpenCL kernel. The module is reset only when the OpenCL kernel is first loaded onto the FPGA, either via Intel FPGA SDK for OpenCL `program` utility or the `clCreateProgramwithBinary` host function. In particular, if the host restarts a kernel via successive `clEnqueueNDRangeKernel` or `clEnqueueTask` invocations, the associated RTL modules will not reset in between these restarts.

The following steps outline the process of setting the kernel clock frequency:

1. The Intel Quartus Prime software's Fitter applies an aggressive constraint on the kernel clock.
2. The Intel Quartus Prime software's Timing Analyzer performs static timing analysis to determine the frequency that the Fitter actually achieves.
3. The phase-locked loop (PLL) that drives the kernel clock sets the frequency determined in Step 2 to be the kernel clock frequency.

Optionally, an RTL module can access a system-wide clock that runs at twice the frequency of the OpenCL kernel clock. This system-wide clock can be connected to an input signal of the RTL module by including an AVALON element of type `clock2x`. These two clocks do not have a defined phase-relationship.

12.1.1.6.1. Intel Stratix® 10 Design-Specific Reset Requirements for Stall-Free and Stallable RTL Modules

When creating an RTL module for Intel Stratix® 10 OpenCL designs, ensure that the module satisfies specific logic reset requirements.

Reset Requirements for Stall-Free RTL Modules

A stall-free RTL module is a fixed-latency module for which the Intel FPGA SDK for OpenCL Offline Compiler can optimize away stall logic.

- When creating a stall-free RTL module for a Intel Stratix 10 design, use synchronous clear signals only.
- After **deassertion** of the reset signal to the stall-free RTL module, the module must be operational within 15 clock cycles. If the reset signal is pipelined within the module, this requirement limits the reset pipelining to no more than 15 stages.

Reset Requirements for Stallable RTL Modules

A stallable RTL module has a variable latency, and it relies on backpressured input and output interfaces to function correctly.

- When creating a stallable RTL module for a Intel Stratix 10 design, use synchronous clear signals only.
- After **assertion** of the reset signal to the stallable RTL module, the module must deassert its `oready` and `ovalid` interface signals within 40 clock cycles.
- After **deassertion** of the reset signal to the stallable RTL module, the module must be fully operational within 40 clock cycles. The module signals its readiness by asserting the `oready` interface signal.

Related Information

Stall-Free RTL on page 145

12.1.1.7. XML Syntax of an RTL Module

This section provides the syntax of a simple XML specification file for an RTL module that implements double-precision square root function. The RTL module is implemented in VHDL with a Verilog wrapper.

The following XML specification file is for an RTL module named `my_fp_sqrt_double` (line 2) that implements an OpenCL helper function named `my_sqrtfd` (line 2).

```
 1: <RTL_SPEC>
 2:   <FUNCTION name="my_sqrtfd" module="my_fp_sqrt_double">
 3:     <ATTRIBUTES>
 4:       <IS_STALL_FREE value="yes"/>
 5:       <IS_FIXED_LATENCY value="yes"/>
 6:       <EXPECTED_LATENCY value="31"/>
 7:       <CAPACITY value="1"/>
 8:       <HAS_SIDE_EFFECTS value="no"/>
 9:       <ALLOW_MERGING value="yes"/>
10:     </ATTRIBUTES>
11:     <INTERFACE>
12:       <AVALON port="clock" type="clock"/>
13:       <AVALON port="resetn" type="resetn"/>
14:       <AVALON port="ivalid" type="ivalid"/>
15:       <AVALON port="iready" type="iready"/>
16:       <AVALON port="ovalid" type="ovalid"/>
17:       <AVALON port="oready" type="oready"/>
18:       <INPUT port="datain" width="64"/>
19:       <OUTPUT port="dataout" width="64"/>
20:     </INTERFACE>
21:     <C_MODEL>
22:       <FILE name="c_model.cl" />
23:     </C_MODEL>
24:     <REQUIREMENTS>
25:       <FILE name="my_fp_sqrt_double_s5.v" />
26:       <FILE name="fp_sqrt_double_s5.vhd" />
27:     </REQUIREMENTS>
28:     <RESOURCES>
29:       <ALUTS value="2057"/>
30:       <FFS value="3098"/>
31:       <RAMS value="15"/>
32:       <MLABS value="43"/>
33:       <DSPS value="1.5"/>
34:     </RESOURCES>
35:   </FUNCTION>
36: </RTL_SPEC>
```

Table 5. Elements and Attributes in the XML Specification File

XML Element	Description
`RTL_SPEC`	Top-level element in the XML specification file. There can only be one such top-level element in the file. In this example, the name RTL_SPEC is historic and carries no file-specific meaning.
`FUNCTION`	Element that defines the OpenCL function that the RTL module implements. The `name` attribute within the `FUNCTION` element specifies the function's name. You may have multiple `FUNCTION` elements, each declaring a different function that you can call from the OpenCL kernel. The same RTL module can implement multiple functions by specifying different parameters.
`ATTRIBUTES`	Element containing other XML elements that describe various characteristics (for example, latency) of the RTL module. The example RTL module takes one `PARAMETER` setting named `WIDTH`, which has a value of 32. Refer to Table 6 on page 151 for more details other `ATTRIBUTES`-specific elements. *Note:* If you create multiple OpenCL helper functions for different modules, or use the same RTL module with different `PARAMETER` settings, you must create a separate `FUNCTION` element for each function.
`INTERFACE`	Element containing other XML elements that describe the RTL module's interface. The example XML specification file shows the Avalon-ST interface signals that every RTL module must provide (that is, `clock`, `resetn`, `ivalid`,

continued...

XML Element	Description
	`iready`, `ovalid`, and `oready`). The signal names must match the ones specified in the `.xml` file. An error will occur during library creation if a signal name is inconsistent.
`C_MODEL`	Element specifying one or more files that implement OpenCL C model for the function. The model is used only during emulation. However, the `C_MODEL` element and the associated file(s) must be present when you create the library file.
`REQUIREMENTS`	Element specifying one or more RTL resource files (that is, `.v`, `.sv`, `.vhd`, `.hex`, and `.mif`). The specified paths to these files are relative to the location of the XML specification file. Each RTL resource file becomes part of the associated Platform Designer component that corresponds to the entire OpenCL system. *Note:* The Intel FPGA SDK for OpenCL library feature does not support `.qip` files. An Intel FPGA SDK for OpenCL Offline Compiler error will occur if you compile an OpenCL kernel while using a library that includes an unsupported resource file type.
`RESOURCES`	Optional element specifying the FPGA resources that the RTL module uses. If you do not specify this element, the FPGA resources that the RTL module uses will default to zero.

12.1.1.7.1. XML Elements for ATTRIBUTES

In the XML specification file of the RTL module within an Intel FPGA SDK for OpenCL library, there are XML elements under ATTRIBUTES that you can specify to set the module's characteristics.

Table 6. XML Elements Associated with the ATTRIBUTES Element in the XML Specification File of an RTL Module

Attention: Except for `IS_STALL_FREE` and `EXPECTED_LATENCY`, all elements have safe values. If you are unsure which value you should specify for an attribute, set it to the safe value. Compiling your kernel with a library that uses safe values will result in functional hardware. However, the hardware might be larger than the actual size.

XML Element	Description
`IS_STALL_FREE`	Instructs the Intel FPGA SDK for OpenCL Offline Compiler to remove all stall logic around the RTL module. Set `IS_STALL_FREE` to `"yes"` to indicate that the module neither generates stalls internally nor can it properly handle incoming stalls. The module simply ignores its stall input. If you set `IS_STALL_FREE` to `"no"`, the module must properly handle all stall and valid signals. *Note:* If you set `IS_STALL_FREE` to `"yes"`, you must also set `IS_FIXED_LATENCY` to `"yes"`. Also, if the RTL module has an internal state, it must properly handle `ivalid=0` inputs. An incorrect `IS_STALL_FREE` setting will lead to incorrect results in hardware.
`IS_FIXED_LATENCY`	Indicates whether the RTL module has a fixed latency. Set `IS_FIXED_LATENCY` to `"yes"` if the RTL module always takes a known number of clock cycles to compute its output. The value you assign to the `EXPECTED_LATENCY` element specifies the number of clock cycles. The safe value for `IS_FIXED_LATENCY` is `"no"`. When you set `IS_FIXED_LATENCY="no"`, the `EXPECTED_LATENCY` value must be at least 1. *Note:* For a given module, you may set `IS_FIXED_LATENCY` to `"yes"` and `IS_STALL_FREE` to `"no"`. Such a module produces its output in a fixed number of clock cycles and handles stall signals properly.
`EXPECTED_LATENCY`	Specifies the expected latency of the RTL module.

continued...

XML Element	Description
	If you set IS_FIXED_LATENCY to "yes", the EXPECTED_LATENCY value indicates the number of pipeline stages inside the module. In this case, you must set this value to be the exact latency of the module. Otherwise, the offline compiler will generate incorrect hardware. For a module with variable latency, the offline compiler balances the pipeline around this module to the EXPECTED_LATENCY value that you specify. For modules that can stall and require use of signals such as iready, the EXPECTED_LATENCY value must be set to at least 1. The specified value and the actual latency might differ, which might affect the number of stalls inside the pipeline. However, the resulting hardware will be correct.
CAPACITY	Specifies the number of multiple inputs that this module can process simultaneously. You must specify a value for CAPACITY if you also set IS_STALL_FREE="no" and IS_FIXED_LATENCY="no". Otherwise, you do not need to specify a value for CAPACITY. If CAPACITY is strictly less than EXPECTED_LATENCY, the offline compiler will automatically insert capacity-balancing FIFO buffers after this module when necessary. The safe value for CAPACITY is 1.
HAS_SIDE_EFFECTS	Indicates whether the RTL module has side effects. Modules that have internal states or communicate with external memories are examples of modules with side effects. Set HAS_SIDE_EFFECTS to "yes" to indicate that the module has side effects. Specifying HAS_SIDE_EFFECTS to "yes" ensures that optimization efforts do not remove calls to modules with side effects. Stall-free modules with side effects (that is, IS_STALL_FREE="yes" and HAS_SIDE_EFFECTS="yes") must properly handle ivalid=0 input cases because the module might receive invalid data occasionally. The safe value for HAS_SIDE_EFFECTS is "yes".
ALLOW_MERGING	Instructs the offline compiler to merge multiple instances of the RTL module. Set ALLOW_MERGING to "yes" to allow merging of multiple instances of the module. Intel recommends setting ALLOW_MERGING to "yes". The safe value for ALLOW_MERGING is "no". *Note:* Marking the module with HAS_SIDE_EFFECTS="yes" does not prevent merging.
PARAMETER	Specifies the value of an RTL module parameter. PARAMETER attributes: • name—Specifies the name of the RTL module parameter. • value—Specifies a decimal numeric value for the parameter. • type—Specifies a system parameter the value of which will be used as the RTL module parameter value. You can use the following system parameter name with the type attribute: — bspaddresswidth—Specifies the Avalon memory bus width required to address the memory range configured for OpenCL global memory in the board support package. *Note:* The value for an RTL module parameter can be specified using either a value or a type attribute.

12.1.1.7.2. XML Elements for INTERFACE

In the XML specification file of the RTL module within an Intel FPGA SDK for OpenCL library, there are XML elements under INTERFACE that you can define to specify aspects of the RTL module's interface (for example, Avalon-ST interface).

Table 7. Mandatory XML Elements Associated with the INTERFACE Element in the XML Specification File of an RTL Module

XML Element	Description
INPUT	Specifies the input parameter of the RTL module.

continued...

XML Element	Description
	INPUT attributes: • port—Specifies the port name of the RTL module. • width—Specifies the width of the port in bits. AOCL only supports widths that correspond to OpenCL data types (that is, 8 (uchar), 16, 32, 64, 128, 256, 512, and 1024 bits (long16)). *Note:* Size of a *type*3 vector is 4 x sizeof(*type*), giving the impression that valid sizes of 24, 48, 96, and 192 bits are unsupported. The input parameters are concatenated to form the input stream. Aggregate data structures such as structs and arrays are not supported as input parameters.
OUTPUT	Specifies the output parameter of the RTL module. OUTPUT attributes: • port—Specifies the port name of the RTL module. • width—Specifies the width of the port in bits. The SDK only supports widths that correspond to OpenCL data types (that is, 8 (uchar), 16, 32, 64, 128, 256, 512, and 1024 bits (long16)). *Note:* Size of *type*3 vector is 4 x sizeof(*type*), giving the impression that valid sizes of 24, 48, 96, and 192 bits are unsupported. The return value from the input stream is sent out via the output parameter on the output stream. Aggregate data structures such as structs and arrays are not supported as input parameters.

If your RTL module communicates with external memory, you need to include additional XML elements:

```
<MEM_INPUT port="m_input_A" access="readonly"/>
<MEM_INPUT port="m_input_sum" access ="readwrite"/>
<AVALON_MEM port="avm_port0" width="512" burstwidth="5" optype="read"
buffer_location=""/>
```

Table 8. Additional XML Elements to Support External Memory Access

XML Element	Description
MEM_INPUT	Describes a pointer input to the RTL module. MEM_INPUT attributes: • port—Specifies the name of the pointer input. • access—Specifies to the Intel FPGA SDK for OpenCL Offline Compiler how the RTL module will use this pointer. Valid access values are readonly and readwrite. If the RTL module only writes with this pointer, assign readwrite to access. Because all pointers to external memory must be 64 bits, there is no width attribute associated with MEM_INPUT.
AVALON_MEM	Declares the Avalon-MM interface for your RTL module.

continued...

XML Element	Description
	AVALON_MEM attributes: • `port`—Specifies the root of the corresponding port names in the RTL module. For example, if `port` has a value of `avm_port0_`, the names of all Avalon-MM interface ports for the RTL module will start with `avm_port0_`. • `width`—Specifies the data width, which must match the corresponding width value in the accelerator board's `board_spec.xml` file. Within the `board_spec.xml` file, the width value is specified in the `interface` element under `global_mem`. For more information, refer to the *global_mem* section under *XML Elements, Attributes, and Parameters in the board_spec.xml File* in the *Intel FPGA SDK for OpenCL Custom Platform Toolkit User Guide*. • `burstwidth`—Specifies the number of bits required to represent burst size. Use `burstwidth = log(maxburst) +1` to calculate the burst size, where `maxburst` is the corresponding maximum burst size specified in the `board_spec.xml` file. For example, if `maxburst=16`, `burstwidth=5`. • `optype`—Specifies either the Avalon-MM port is reading (`read`) or writing (`write`) from external memory. You can only assign either `read` or `write` to `optype`. • `buffer_location`—Supports heterogeneous memory. Leave this attribute blank because the heterogeneous memory compilation flow is currently untested.

For the AVALON_MEM element defined in the code example above, the corresponding RTL module ports are as follows:

```
output         avm_port0_enable,
input  [511:0] avm_port0_readdata,
input          avm_port0_readdatavalid,
input          avm_port0_waitrequest,
output [31:0]  avm_port0_address,
output         avm_port0_read,
output         avm_port0_write,
input          avm_port0_writeack,
output [511:0] avm_port0_writedata,
output [63:0]  avm_port0_byteenable,
output [4:0]   avm_port0_burstcount,
```

There is no assumed correspondence between pointers that you specify with MEM_INPUT and the Avalon-MM interfaces that you specify with AVALON_MEM. An RTL module can use a single pointer to address zero to multiple Avalon-MM interfaces.

Related Information

XML Elements, Attributes, and Parameters in the board_spec.xml File: global_mem

12.1.1.7.3. XML Elements for RESOURCES

In the XML specification file of the RTL module within an Intel FPGA SDK for OpenCL library, there are optional elements under RESOURCES that you can define to specify the FPGA resource utilization of the module. If you do not specify a particular element, it will have a default value of zero.

Table 9. XML Elements Associated with the RESOURCES Element in the XML Specification File of an RTL Module

XML Element	Description
ALUTS	Specifies the number of combinational adaptive look-up tables (ALUTs) that the module uses.
FFS	Specifies the number of dedicated logic registers that the module uses.
RAMS	Specifies the number of block RAMs that the module uses.
DSPS	Specifies the number of digital signal processing (DSP) blocks that the module uses.
MLABS	Specifies the number of memory logic arrays (MLABs) that the module uses. This value is equal to the number of adaptive logic modules (ALMs) that is used for memory divided by 10 because each MLAB consumes 10 ALMs.

12.1.1.8. Interaction between RTL Module and External Memory

Allow your RTL module to interact with external memory only if the interaction is necessary and unavoidable.

Important: The preferred method for having your RTL module interact with external memory is to have the OpenCL kernel access global memory and then feed that memory content to the RTL module. For operations like reading from and writing to external memory on every kernel invocation, instruct the OpenCL kernel to perform the operation. To do so, you can create an OpenCL helper function for the OpenCL kernel in the same Intel FPGA SDK for OpenCL library as the RTL module.

The following examples demonstrate how to structure code in an RTL module for easy integration into an OpenCL library:

Table 10. Example Code in an RTL Module that Interacts with External Memory

Complex RTL Module	Simplified RTL Module

`// my_rtl_fn does:`
`// out_ptr[idx] = fn(in_ptr[idx])`
`my_rtl_fn (in_ptr, out_ptr,idx);`
	`int in_value = in_ptr[idx];`
	`// my_rtl_fn now does: out = fn(in)`
	`int out_value = my_rtl_fn (in_value);`
	`out_ptr[idx] = out_value;`

The complex RTL module on the left reads a value from external memory, performs a scalar function on the value, and then writes the value back to global memory. Such an RTL module is difficult to describe when you integrate it into an OpenCL library. In addition, this RTL module is harder to verify and causes very conservative pointer analysis in the Intel FPGA SDK for OpenCL Offline Compiler.

The simplified RTL module on the right provides the same overall functionality as the complex RTL module. However, the simplified RTL module only performs a scalar-to-scalar calculation without connecting to global memory. Integrating this simplified RTL module into the OpenCL library makes it much easier for the offline compiler to analyze the resulting OpenCL kernel.

There are times when an RTL module requires an Avalon-MM port to communicate with external memory. This Avalon-MM port connects to the same arbitration network to which all other global load and store units in the OpenCL kernels connect.

If an RTL module receives a memory pointer as an argument, the offline compiler enforces the following memory model:

- If an RTL module writes to a pointer, nothing else in the OpenCL kernel can read from or write to this pointer.
- If an RTL module reads from a pointer, the rest of the OpenCL kernel and other RTL modules may also read from this pointer.
- You may set the `access` field of the `MEM_INPUT` attribute to specify how the RTL module uses the memory pointer. Ensure that you set the value for access correctly because there is no way to verify the value.

12.1.1.9. Order of Threads Entering an RTL Module

Do not assume that threads entering an RTL module follow a defined order. In addition, an RTL module can reorder threads. As a result, thread 0 does not necessarily enter the module before thread 1.

12.1.1.10. OpenCL C Model of an RTL Module

Each RTL module within an OpenCL library must have an OpenCL C model. The OpenCL C model verifies the overall OpenCL system during emulation.

Example OpenCL C model file for a square root function:

```
double my_sqrtfd (double a)
{
    return sqrt(a);
}
```

Intel recommends that you emulate your OpenCL system. If you decide not to emulate your OpenCL system, create an empty function with a name that matches the function name you specified in the XML specification file.

Related Information

XML Syntax of an RTL Module on page 149

12.1.1.11. Potential Incompatibility between RTL Modules and Partial Reconfiguration

When creating an OpenCL library using RTL modules, you might encounter Partial Reconfiguration-related issues.

Consider a situation where you create and verify your library on a device that does not support Partial Reconfiguration (PR). If a library user then uses the library's RTL module inside a PR region, the module might not function correctly after PR.

To ensure that the RTL modules function correctly on a device that uses PR:

- The RTL modules do not use memory logic array blocks (MLABs) with initialized content.
- The RTL modules do not make any assumptions regarding the power-up values of any logic.

For complete PR coding guidelines, refer to Creating a Partial Reconfiguration Design in the *Partial Reconfiguration User Guide*.

12.1.2. Packaging an OpenCL Helper Function File for an OpenCL Library

Before creating an OpenCL library file, package each OpenCL source file with helper functions into a `.aoco` file. Unlike RTL modules, you do not need to create an XML specification file.

In general, you do not need to create a library to share helper functions written in OpenCL. You can distribute a helper function in source form (for example, `<shared_file>.cl`) and then insert the line `#include "<shared_file>.cl"` in the OpenCL kernel source code.

Consider creating a library under the following circumstances:

- The helper functions are in multiple files and you want to simplify distribution.
- You do not want to expose the helper functions' source code.

  The helper functions are stored as LLVM IR, an assembly-like language, without comments inside the associated library.

Hardware generation is not necessary for the creation of a `.aoco` file. Compile the OpenCL source file using the `-c` offline compiler command option.

Note: A library can only include OpenCL helper functions. The Intel FPGA SDK for OpenCL Offline Compiler will issue an error message if the library contains OpenCL kernels.

- To package an OpenCL source file into a `.aoco` file, invoke the following command: `aoc -c -shared <OpenCL_source_file_name>.cl -o <OpenCL_object_file_name>.aoco`

  where the `-shared` offline compiler command option instructs the compiler to create a `.aoco` file that is suitable for inclusion into an OpenCL library.

Related Information

- Packaging Multiple Object Files into a Library File on page 160
- Specifying an OpenCL Library when Compiling an OpenCL Kernel on page 160

12.1.3. Packaging an RTL Component for an OpenCL Library

Before creating an OpenCL library file, package each RTL component into a `.aoco` file.

Hardware generation is not necessary for the creation of a `.aoco` file. Compile the OpenCL source file using the `-c` Intel FPGA SDK for OpenCL Offline Compiler command option.

- To package an RTL component into a `.aoco` file, invoke the following command: `aoc -c <RTL component description file name>.xml -o <RTL object file name>.aoco`

Related Information

- Packaging Multiple Object Files into a Library File on page 160
- Verifying the RTL Modules on page 159
- Specifying an OpenCL Library when Compiling an OpenCL Kernel on page 160

12.1.3.1. Restrictions and Limitations in RTL Support for the Intel FPGA SDK for OpenCL Library Feature

The Intel FPGA SDK for OpenCL supports the use of RTL modules in an OpenCL library with some restrictions and limitations.

When creating your RTL module, ensure that it operates within the following restrictions:

- An RTL module must use a single input Avalon-ST interface. That is, a single pair of `ready` and `valid` logic must control all the inputs.

  You have the option to provide the necessary Avalon-ST ports but declare the RTL module as stall-free. In this case, you do not have to implement proper stall behavior because the Intel FPGA SDK for OpenCL Offline Compiler creates a wrapper for your module. Refer to *XML Syntax of an RTL Module* and *Using an OpenCL Library that Works with Simple Functions (Example 1)* for more syntax and usage information, respectively.

  Note: You must handle `ivalid` signals properly if your RTL module has an internal state. Refer to *Stall-Free RTL* for more information.
- The RTL module must work correctly regardless of the kernel clock frequency.
- Data input and output sizes must match valid OpenCL data types, from 8 bits for `char` to 1024 bits for `long16`.

  For example, if you work with 24-bit values inside an RTL module, declare inputs to be 32 bits and declare function signature in the SDK's library header file to accept the `uint` data type. Then, inside the RTL module, accept the 32-bit input but discard the top 8 bits.
- RTL modules cannot connect to external I/O signals. All input and output signals must come from an OpenCL kernel.
- An RTL module must have a `clock` port, a `resetn` port, and Avalon-ST input and output ports (that is, `ivalid`, `ovalid`, `iready`, `oready`). Name the ports as specified here.
- RTL modules that communicate with external memory must have Avalon Memory-Mapped (Avalon-MM) port parameters that match the corresponding Custom Platform parameters. The offline compiler does not perform any width or burst adaptation.
- RTL modules that communicate with external memory must behave as follows:
  - They cannot burst across the burst boundary.
  - They cannot make requests every clock cycle and stall the hardware by monopolizing the arbitration logic. An RTL module must pause its requests regularly to allow other load or store units to execute their operations.
- RTL modules cannot act as stand-alone OpenCL kernels. RTL modules can only be helper functions and be integrated into an OpenCL kernel during kernel compilation.
- Every function call that corresponds to RTL module instantiation is completely independent of other instantiations. There is no hardware sharing.
- Do not incorporate kernel code (that is, functions marked as `kernel`) into a `.aoclib` library file. Incorporating kernel code into the library file causes the offline compiler to issue an error message. You may incorporate helper functions into the library file.

- An RTL component must receive all its inputs at the same time. A single `ivalid` input signifies that all inputs contain valid data.
- The SDK does not support I/O RTL modules.
- You can only set RTL module parameters in the *<RTL module description file name>*`.xml` specification file, not the OpenCL kernel source file. To use the same RTL module with multiple parameters, create a separate `FUNCTION` tag for each parameter combination.

Currently, the SDK's RTL module support for the library feature has the following limitations:

- You can only pass data inputs to an RTL module by value via the OpenCL kernel code. Do not pass data inputs to an RTL module via pass-by reference, structs, or channels. In the case of channel data, extract the data from the channel first and then pass the extracted the scalar data to the RTL module.

  Note: Passing data inputs to an RTL module via pass-by reference or structs will cause a fatal error to occur in the offline compiler.
- The debugger (for example, GDB for Linux) cannot step into a library function during emulation. In addition, optimization and area reports will not include code line numbers beside the library functions.
- Names of RTL module source files cannot conflict with the file names of Intel FPGA SDK for OpenCL Offline Compiler IP. Both the RTL module source files and the offline compiler IP files are stored in the *<kernel file name>*`/system/synthesis/submodules` directory. Naming conflicts will cause existing offline compiler IP files in the directory to be overwritten by the RTL module source files.
- The SDK does not support `.qip` files. You must manually parse nested `.qip` files to create a flat list of RTL files.
- It is very difficult to debug an RTL module that works correctly on its own but works incorrectly as part of an OpenCL kernel. Double check all parameters under the `ATTRIBUTES` element in the *<RTL module description file name>*`.xml` file.
- All offline compiler area estimation tools assume that RTL module area is 0. The SDK does not currently support the capability of specifying an area model for RTL modules.
- RTL modules cannot access a 2x clock that is in-phase with the kernel clock and at twice the kernel clock frequency.

Related Information

- XML Syntax of an RTL Module on page 149
- Using an OpenCL Library that Works with Simple Functions (Example 1) on page 164
- Stall-Free RTL on page 145

12.1.4. Verifying the RTL Modules

The creator of an OpenCL library is responsible for verifying the RTL modules within the library, both as stand-alone entities and as part of an OpenCL system.

1. Verify each RTL module using standard hardware verification methods.
2. Modify one of Intel FPGA SDK for OpenCL library design examples to test your RTL modules inside the overall OpenCL system.

   This testing step is critical to prevent library users from encountering hardware problems.

   It is crucial that you set the values for the ATTRIBUTES elements in the XML specification file correctly. Because you cannot simulate the entire OpenCL system, you will likely not discover problems caused by interface-level errors until hardware runs.
3. *Note:* The Intel FPGA SDK for OpenCL `library` utility performs consistency checks on the XML specification file and source files, with some limitations.

   Invoke the `aocl library [<command option>]` command.

   - For a list of supported *<command options>*, invoke the `aocl library` command.
   - The `library` utility does not detect errors in values assigned to elements within the ATTRIBUTES, MEM_INPUT, and AVALON_MEM elements in the XML specification file.
   - The `library` utility does not detect RTL syntax errors. You must check the *<your_kernel_filename>*`/quartus_sh_compile.log` file for RTL syntax errors. However, parsing the errors might be time consuming.

12.1.5. Packaging Multiple Object Files into a Library File

After creating the `.aoco` files that you want to include into an OpenCL library, package them into a library file by invoking the Intel FPGA SDK for OpenCL `library` utility command option.

- To package multiple object files into a single library file, invoke the following command: `aocl library create -o <library file name>.aoclib <object file 1>.aoco [<object file 2>.aoco ... <object file N>.aoco]`

  The `aocl library` utility command creates a *<library file name>*`.aoclib` library file, which includes the `.aoco` object files you specify in the command. A library file may contain both RTL-based object files and OpenCL-based object files.

12.1.6. Specifying an OpenCL Library when Compiling an OpenCL Kernel

To use an OpenCL library in an OpenCL kernel, specify the library file name and directory when you compile the kernel.

Important: Using a library does not reduce kernel compilation time.

- To specify an OpenCL library to the Intel FPGA SDK for OpenCL Offline Compiler, invoke the following command: `aoc -l <library_file_name>.aoclib [-L <library directory>] <kernel file name>.cl`

  where the `-l <library_file_name>.aoclib` command option specifies the library file name, and the `-L <library directory>` command option specifies the directory containing the library files.

  You may include multiple instances of `-l <library file name>` and `-L <library directory>` in the offline compiler command.

For example, if you create a library that includes the functions `my_div_fd()`, `my_sqrtfd()`, and `myrsqrtfd()`, the OpenCL kernel code might resemble the following:

```
#include “lib_header.h”

kernel void test_lib (
    global double * restrict in,
    global double * restrict out,
    int N) {
        int i = get_global_id(0);
        for (int k =0; k < N; k++) {
            double x = in[i*N + k];
            out[i*N + k] = my_divfd
                (my_rsqrtfd(x),
                my_sqrtfd(my_rsqrtfd (x)));
        }
}
```

Note: Library-related lines are highlighted in bold.

The corresponding `lib_header.h` file might resemble the following:

```
double my_sqrtfd (double x);
double my_rsqrtfd(double x);
double my_divfd(double a, double b);
```

12.1.7. Debugging Your OpenCL Library Through Simulation (Preview)

The Intel FPGA SDK for OpenCL simulator assesses the functionality of your OpenCL library.

The Intel FPGA SDK for OpenCL simulator generates a `.aocx` file that runs on an x86-64 Windows or a Linux host. This feature allows you to simulate the functionality of your kernel and iterate on your design without needing to compile your library to hardware and running on the FPGA each time.

Use the simulator when you want insight into the dynamic performance of your OpenCL library and more information about the functional correctness of your OpenCL library than emulation or the OpenCL reporting tools provide.

The simulator is cycle accurate, has a netlist identical to generate hardware, and can provide full waveforms for debugging. View the waveforms with Mentor Graphics* ModelSim* software.

12.1.7.1. Compiling a Library for Simulation (-march=simulator)

To compile an OpenCL library for simulation, include the `-march=simulator` option in your `aoc` command. To enable collecting the waveform during the simulation, include the `-ghdl` option in your `aoc` command.

- Before you perform library simulation, perform the following tasks:
  - Install a Custom Platform from your board vendor for your FPGA accelerator boards.
  - Verify that the environment variable *QUARTUS_ROOTDIR_OVERRIDE* points to Intel Quartus Prime Pro Edition software installation folder.
- To simulate library on Windows systems, you need the Microsoft linker and additional compilation time libraries. Verify that the *PATH* environment variable setting includes all the paths described in the *Setting the Intel FPGA SDK for OpenCL Pro Edition User Environment Variables* section of the *Intel FPGA SDK for OpenCL Pro Edition Getting Started Guide*.

  The *PATH* environment variable setting must include the path to the `LINK.EXE` file in Microsoft Visual Studio.
- Ensure that your *LIB* environment variable setting includes the path to the Microsoft compilation time libraries.

  The compilation time libraries are available with Microsoft Visual Studio.
- Verify that the *LD_LIBRARY_PATH* environment variable setting includes all the paths described in the *Setting the Intel FPGA SDK for OpenCL Pro Edition User Environment Variables* section in the *Intel FPGA SDK for OpenCL Pro Edition Getting Started Guide*.
- To compile a simulation that targets a specific board, invoke the `aoc -march=simulator -ghdl -board=<board_name> <your_kernel_filename>.cl` command.
- For Linux systems, the Intel FPGA SDK for OpenCL Offline Compiler offers symbolic debug support for the debugger.

  The offline compiler debug support allows you to pinpoint the origins of functional errors in your kernel source code.

12.1.7.2. Simulating Your OpenCL Library

If you want to view the waveforms generated during simulation, you must install and configure Mentor Graphics ModelSim software.

Important: If you want to run the emulator and the simulator from the same terminal or command prompt session, unset the emulator environment variable (`CL_CONTEXT_EMULATOR_DEVICE_INTELFPGA`) before trying to run the simulator. If you do not unset the emulator environment variable, your simulation fails with errors.

You can also run the emulator and simulator from separate terminal or command prompt sessions.

To run your OpenCL library through the simulator:

1. Run the utility command `aocl linkflags` to find out which libraries are necessary for building a host application. The software lists the libraries for both emulation and regular kernel compilation flows.
2. Build a host application and link it to the libraries from Step 1.

   Tip: To emulate multiple devices alongside other OpenCL SDKs, link your host application to the Khronos ICD Loader Library *before* you link it to the host runtime libraries. Link the host application to the ICD Loader Library by modifying the `Makefile` for the host application. For more information, see *Linking Your Host Application to the Khronos ICD Loader Library*.
3. If necessary, move the `.aocx` file to a location where the host can find easily, preferably the current working directory.
4. Set the `CL_CONTEXT_MPSIM_DEVICE_INTELFPGA` environment variable to enable the simulation device:
   - Windows:

     ```
     set CL_CONTEXT_MPSIM_DEVICE_INTELFPGA=1
     ```
   - Linux:

     ```
     env CL_CONTEXT_MPSIM_DEVICE_INTELFPGA=1
     ```

   Remember: When the environment variable *CL_CONTEXT_MPSIM_DEVICE_INTELFPGA* is set, only the simulation devices are available. That is, access to physical boards and the emulation device is disabled.

   You might need to set `CL_CONTEXT_COMPILER_MODE_INTELFPGA=3` if the host program cannot find the simulator device.
5. Run your host program.

   To debug your host code and device, you can run your host code in gdb or Eclipse.

   Running the host program gives you a a waveform file, `vsim.wif`, that you can view in ModelSim software as your host code executes. The `vsim.wif` file is written to the same directory that you run your host program from.
6. If you change your host or kernel program and you want to test it, only recompile the modified host or kernel program and then rerun simulation.

12.1.7.3. Limitations of the Simulator

Review the limitations of the simulator to troubleshoot problems you might have when attempting to run a simulation.

Windows Compilation Fails - Host Program Reports Corrupt `.aocx` file

During the compilation of the `device.cl` file, your directory path is likely too long. Use the `-o` compiler option to output your compilation results to a shorter path.

A socket=-11 Error Is Logged to transcript.log

If you receive the following error message, you are mixing resources from ModelSim - Intel FPGA Edition and ModelSim SE:

```
Message: "src/hls_cosim_ipc_socket.cpp:202: void IPCSocketMaster::connect():
Assertion `sockfd != -1 && "IPCSocketMaster::connect() call to accept()
failed"' failed."
```

An example of mixing ModelSim resources is compiling a device with ModelSim SE and then running the host program in ModelSim - Intel FPGA Edition.

Running the Host Program Generates a Segmentation Fault

If you receive a segmentation fault when you run your host program, you might be running the emulator and the simulator from the same terminal or command prompt sessions. Remember to unset emulator environment variables before trying to run the simulator.

Try to avoid compiling your device and your host program in the same terminal or command prompt sessions. By using separate sessions, you can avoid possible environment variable conflicts.

12.1.8. Using an OpenCL Library that Works with Simple Functions (Example 1)

Intel provides an OpenCL library design example of a simple kernel that uses a library containing RTL implementations of three double-precision functions: `sqrt`, `rsqrt`, and `divide`.

The `example1.tgz` tar ball includes a library, a kernel, and a host system. The `example1.cl` kernel source file includes two kernels. The kernel `test_lib` uses library functions; the kernel `test_builtin` uses built-in functions. The host runs both kernels and then compares their outputs and runtimes. Intel recommends that you use the same strategy to verify your own library functions.

To compile this design example, perform the following tasks:

1. Obtain `example1.tgz` from the OpenCL Design Examples web page.
2. Unpack it into a local directory.
3. Follow the instructions in the `README.html` file, which is located in the top-level of the unpacked example.
   When you run the compiled host program, it should produce the following output:

```
Loading example1.aocx ...
Create buffers
Generate random data for conversion...
Enqueuing both library and builtin in kernels 4 times with global size 65536
Kernel computation using library function took 5.35333 seconds
Kernel computation using built-in function took 5.39949 seconds
Reading results to buffers...
Checking results...
Library function throughput is within 5% of builtin throughput.
PASSED
```

Related Information

OpenCL Design Examples page

12.1.9. Using an OpenCL Library that Works with External Memory (Example 2)

Intel provides an OpenCL library design example of a simple kernel that uses a library containing two RTL modules that communicate with global memory.

The `example2.tgz` tar ball includes a library, a kernel, and a host system. In this example, the RTL code that communicates with global memory is Custom Platform- or Reference Platform-dependent. Ensure that the compilation targets the board that corresponds to the Stratix V Network Reference Platform.

Intel generated the RTL modules `copyElement()` and `sumOfElements()` using the Intel FPGA SDK for OpenCL Offline Compiler, which explains the extra inputs in the code.

The `example2.cl` kernel source file includes two kernels. The kernel `test6` is an NDRange kernel that calls the `copyElement()` RTL function, which copies data from `B[]` to `A[]` and then stores `global_id+100` in `C[]`. The kernel `test11` is a single work-item kernel that uses an RTL function . The `sumOfElements()` RTL function determines the sum of the elements of `A[]` in range `[i, N]` and then adds the rest to `C[i]`.

Note: First invocations of `sumOfElements(i=0)` will take more time to execute than later invocations.

To compile this design example, perform the following tasks:

1. Obtain the `example2.tgz` from the OpenCL Design Examples web page.
2. Unpack it into a local directory.
3. Follow the instructions in the `README.html` file, which is located in the top-level of the unpacked example.
   When you run the compiled host program, it should produce the following output:

```
Loading example2.aocx ...
Running test6
Launching the kernel test6 with globalsize=128 localSize=16
Loading example2.aocx ...
Running test11
Launching the kernel test11 with globalsize=1 localSize=1
PASSED
```

Related Information

- OpenCL Design Examples page
- Compiling a Kernel for a Specific FPGA Board (-board=<board_name>) on page 107
- Intel FPGA SDK for OpenCL Stratix V Network Reference Platform Porting Guide

12.1.10. OpenCL Library Command-Line Options

Both the Intel FPGA SDK for OpenCL Offline Compiler's set of commands and the SDK utility include options you can invoke to perform OpenCL library-related tasks.

Table 11. Library-Related Intel FPGA SDK for OpenCL Offline Compiler Command Options

Command Option	Description
`-shared`	In conjunction with the `-rtl` command option, compiles an OpenCL source file into an object file (`.aoco`) that you can then include into a library. `aoc -rtl -shared <OpenCL source file name>.cl -o <OpenCL object file name>.aoco`
`-I=<library_directory>`	Adds *<library directory>* to the header file search path. `aocl -I <library_header_file_directory> -l <library_file_name>.aoclib <kernel_file_name>.cl`
`-L=<library directory>`	Adds *<library directory>* to the OpenCL library search path. Space after "-L" is optional. `aoc -l=<library_file_name>.aoclib [-L=<library directory>] <kernel file name>.cl`
`-l=<library_file_name>.aoclib`	Specifies the OpenCL library file (`<library_file_name>.aoclib`). Space after `-l` is optional. `aoc -l=<library_file_name>.aoclib [-L=<library directory>] <kernel file name>.cl`
`-library-debug`	Generates debug output that relates to libraries. Part of the additional output appears in stdout, the other part appears in the `<kernel_file_name>/<kernel_file_name>.log` file. `aoc -l=<library_file_name>.aoclib -library-debug <kernel_file_name>.cl`

Table 12. Intel FPGA SDK for OpenCL Library Utility (aocl library) Command Options

Command Option	Description
`hdl-comp-pkg <XML_specification_ file>.xml`	Packages a single HDL component into a `.aoco` file that you then include into a library. Invoking this command option is similar to invoking `aoc -rtl <XML_specification_file>.xml`. However, the processing time is faster because the `aocl` utility will not perform any environment checks. `aocl library hdl-comp-pkg <XML_specification_ file>.xml -o <output_file>.aoco`
`-rtl <XML_specification_ file>.xml`	Same function as `hdl-comp-pkg <XML_specification_ file>.xml`. `aocl library -rtl <XML_specification_ file>.xml`
`create`	Creates a library file from the `.aoco` files that you created by invoking the `hdl-comp-pkg` utility option or the `aoc -shared` command, and any other `.aoclib` libraries. `aocl library create [-name <library_name>] [-vendor <library_vendor>] [-version <library_version>] [-o <output_file>.aoclib] [.aoco...] [.aoclib...]`

continued...

Command Option	Description
	where `-name`, `-vendor`, and `-version` are optional information strings you can specify and add to the library.
`list <library_name>`	Lists all the RTL components in the library. Currently, this option is not available for use to list OpenCL functions. `aocl library list <library_name>`
`help`	Prints the list of Intel FPGA SDK for OpenCL library utility options and their descriptions on screen. `aocl library help`

12.2. Kernel Attributes for Configuring Local and Private Memory Systems

The Intel FPGA SDK for OpenCL includes kernel attributes that you can include in a kernel to customize the geometry of the local and private memory systems.

Attention: Only apply these local memory kernel attributes to local or private variables.

Table 13. OpenCL Kernel Attributes for Configuring Local Memory

Attribute	Description
`register`	Specifies that the local variable must be implemented in a register.
`memory`	Specifies that the local variable must be implemented in a memory system. Including the memory kernel attribute is equivalent to declaring the local variable with the `__local` qualifier.
`numbanks(N)` *N* is an integer value.	Specifies that the memory system implementing the local variable must have *N* banks, where *N* is a power-of-2 integer value greater than zero.
`bankwidth(N)` *N* is an integer value.	Specifies that the memory system implementing the local variable must have banks that are *N* bytes wide, where *N* is a power-of-2 integer value greater than zero.
`singlepump`	Specifies that the memory system implementing the local variable must be single pumped.
`doublepump`	Specifies that the memory system implementing the local variable must be double pumped.
`numreadports(N)` *N* is an integer value.	Specifies that the memory system implementing the local variable must have *N* read ports, where *N* is an integer value greater than zero.
`numwriteports(N)` *N* is an integer value.	Specifies that the memory system implementing the local variable must have *N* write ports, where *N* is an integer value greater than zero.
`merge("label", "direction")`	Forces two or more variables to be implemented in the same memory system. *label* is an arbitrary string. Assign the same label to all variables that you want to merge. Specify *direction* as either `width` or `depth` to identify whether the memories should be merged width-wise or depth-wise, respectively.
`bank_bits(`b_0`, `b_1`, ..., `b_n`)`	Forces the memory system to split into 2^n banks, with $\{b_0, b_1, ..., b_n\}$ forming the bank-select bits. *Important:* $b_0, b_1, ..., b_n$ must be consecutive, positive integers.

continued...

Attribute	Description
	If you specify the `numbanks(n)` attribute without the `bank_bits` attribute, the bank-select bits default to the least significant bits (that is, 0, 1, ..., log2(numbanks)-1).

Table 14. Code Examples for Memory Attributes

Example Use Case	Syntax
Implements a variable in a register	`int __attribute__((register)) a[12];`
Implements a memory system with eight banks, each with a width of 8 bytes	`int __attribute__((memory, numbanks(8), bankwidth(8)) b[16];`
Implements a double-pumped memory system with one 128-byte wide bank, one write port, and four read ports	`int __attribute__((memory, numbanks(1), bankwidth(128), doublepump, numwriteports(1), numreadports(4)) c[32];`

Related Information

- Improve Kernel Performance by Banking the Local Memory
- Optimize Accesses to Local Memory by Controlling the Memory Replication Factor

12.2.1. Restrictions on the Usage of Variable-Specific Attributes

The Intel FPGA SDK for OpenCL Offline Compiler will error out or issue warnings if it detects unsupported usages of the variable-specific attributes or incorrect memory configurations.

Unsupported usages of variable-specific attributes that cause compilation errors:

- You use the kernel attributes in declarations other than local or private variable declarations (for example, declarations for function parameters, global variable declarations, or function declarations).
- You use the `register` attribute in conjunction with any of the other variable-specific attributes.
- You specify the `numbanks` attribute but not the `bankwidth` attribute in the same variable declaration, or vice versa.
- You include both the `singlepump` and `doublepump` attributes in the same variable declaration.

- You specify the `numreadports` and `numwriteports` attributes without also including the `singlepump` or `doublepump` attribute in the same variable declaration.
- You specify the `numreadports` attribute but not the `numwriteports` attribute in the same variable declaration, or vice versa.
- You specify any of the following attributes without also specifying the `numbanks` and `bankwidth` attributes in the same variable declaration:
  - `numreadports`
  - `numwriteports`
  - `singlepump`
  - `doublepump`

Incorrect memory configurations that cause the offline compiler to issue warnings during compilation:

- The memory configuration that is defined by the variable-specific attributes exceeds the available storage size (for example, specifying eight banks of local memory for an integer variable).

Incorrect memory configurations that cause compilation errors:

- The bank width is smaller than the data storage size (for example, bank width is 2 bytes for an array of 4-byte integers).
- You specify memory configurations for the variables. However, because of compiler restrictions or coding style, the offline compiler implements the variables in the same memory instead of splitting the memory.
- You specify the `register` attribute for a variable. However, because of compiler restrictions or coding style, the offline compiler cannot implement the variable in a register.

12.3. Kernel Attributes for Reducing the Overhead on Hardware Usage

The Intel FPGA SDK for OpenCL includes kernel attributes that you can include in a single work-item kernel to reduce logic utilization and improve kernel performance.These kernel attributes enables the Intel FPGA SDK for OpenCL Offline Compiler to omit the generation of unnecessary hardware to increase efficiency.

12.3.1. Hardware for Kernel Interface

The Intel FPGA SDK for OpenCL Offline Compiler generates hardware around the kernel pipeline. For some OpenCL applications, these interface hardware components are not necessary.

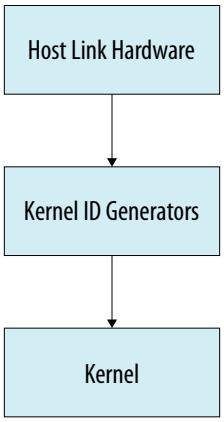
Hardware around the kernel pipeline is necessary for functions such as the following:

- Dispatching IDs for work-items and work-groups
- Communicating with the host regarding kernel arguments and work-group sizes

Figure 20 on page 170 illustrates the hardware that the offline compiler generates when it compiles the following kernel:

```
__kernel void my_kernel(global int* arg)
{
    …
    int sum = 0;
    for(unsigned i = 0; i < n; i++)
    {
        if(sum < m) sum += val;
    }
    *arg = sum;
    …
}
```

Figure 20. Intel FPGA SDK for OpenCL Offline Compiler-Generated Interface Hardware around a Kernel Pipeline



12.3.1.1. Omit Hardware that Generates and Dispatches Kernel IDs

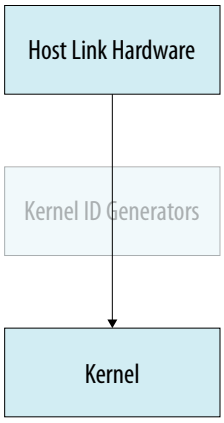
The `max_global_work_dim(0)` kernel attribute instructs the Intel FPGA SDK for OpenCL Offline Compiler to omit logic that generates and dispatches global, local, and group IDs into the compiled kernel.

Semantically, the `max_global_work_dim(0)` kernel attribute specifies that the global work dimension of the kernel is zero. Setting this kernel attribute means that the kernel does not use any global, local, or group IDs. The presence of this attribute in the kernel code serves as a guarantee to the offline compiler that the kernel is a single work-item kernel.

When compiling the following kernel, the offline compiler will generate interface hardware as illustrated in Figure 21 on page 171.

```
channel int chan_in;
channel int chan_out;

__attribute__((max_global_work_dim(0)))
__kernel void plusK (int N, int k) {
    for (int i = 0; i < N; ++i) {
        int data_in = read_channel_intel(chan_in);
        write_channel_intel(chan_out, data_in + k);
    }
}
```

Figure 21. Intel FPGA SDK for OpenCL Offline Compiler-Generated Interface Hardware for a Kernel with the max_global_work_dim(0) Attribute



If your current kernel implementation has multiple work-items but does not use global, local, or group IDs, you can use the `max_global_work_dim(0)` kernel attribute if you modify the kernel code accordingly:

1. Wrap the kernel body in a `for` loop that iterates as many times as the number of work-items.
2. Launch the modified kernel with only one work-item.

12.3.1.2. Omit Communication Hardware between the Host and the Kernel

The `autorun` kernel attribute instructs the Intel FPGA SDK for OpenCL Offline Compiler to omit logic that is used for communication between the host and the kernel. A kernel that uses the `autorun` attribute starts executing automatically before any kernel that the host launches explicitly. In addition, this kernel restarts automatically as soon as it finishes its execution.

The `autorun` kernel attribute notifies the offline compiler that the kernel runs on its own and will not be enqueued by any host.

To leverage the `autorun` attribute, a kernel must meet all of the following criteria:

1. Does not use I/O channels

   Note: Kernel-to-kernel channels are supported.
2. Does not have any arguments
3. Has either the `max_global_work_dim(0)` attribute or the `reqd_work_group_size(`*X*`,`*Y*`,`*Z*`)` attribute

   Note: The parameters of the `reqd_work_group_size(`*X*`,`*Y*`,`*Z*`)` attribute must be divisors of 2^{32}.

As mentioned above, kernels with the `autorun` attribute cannot have any arguments and start executing without the host launching them explicitly. As a result, the offline compiler does not need to generate the logic for communication between the host and the kernel. Omitting this logic reduces logic utilization and allows the offline compiler to apply additional performance optimizations.

A typical use case for the `autorun` attribute is a kernel that reads data from one or more kernel-to-kernel channels, processes the data, and then writes the results to one or more channels. When compiling the kernel, the offline compiler will generate hardware as illustrated in Figure 22 on page 172.

```
channel int chan_in;
channel int chan_out;

__attribute__((max_global_work_dim(0)))
__attribute__((autorun))
__kernel void plusOne () {
    while(1) {
        int data_in = read_channel_intel(chan_in);
        write_channel_intel(chan_out, data_in + 1);
    }
}
```

Figure 22. Single Work-Item Kernel with No Interface Hardware

12.4. Kernel Replication Using the num_compute_units(X,Y,Z) Attribute

You can replicate your single work-item OpenCL kernel by including the `num_compute_units(X,Y,Z)` kernel attribute.

As mentioned in *Specifying Number of Compute Units*, including the `num_compute_units(N)` kernel attribute in your kernel instructs the Intel FPGA SDK for OpenCL Offline Compiler to generate multiple compute units to process data. The `num_compute_unit(N)` attribute instructs the offline compiler to generate *N* identical copies of the kernel in hardware.

Remember: To identify the specific compute unit controlling the data-dependent kernel processing, call the `get_compute_id()` intrinsic function.

Related Information

- Customization of Replicated Kernels Using the get_compute_id() Function on page 173
- Specifying Number of Compute Units on page 30

12.4.1. Customization of Replicated Kernels Using the get_compute_id() Function

To create compute units that are slightly different from one another but share a lot of common code, call the `get_compute_id()` intrinsic function in a kernel that also uses the `num_compute_units (X,Y,Z)` attribute.

Attention: You can only use the `get_compute_id()` function in a kernel that also uses the `autorun` and `max_global_work_dim(0)` kernel attributes.

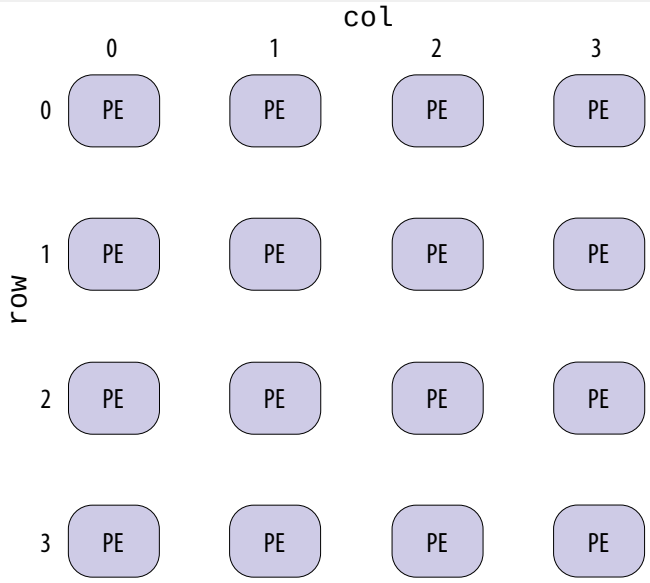
Retrieving compute IDs is a convenient alternative to replicating your kernel in source code and then adding specialized code to each kernel copy. When a kernel uses the `num_compute_units(X,Y,Z)` attribute and calls the `get_compute_id()` function, the Intel FPGA SDK for OpenCL Offline Compiler assigns a unique compute ID to each compute unit. The `get_compute_id()` function then retrieves these unique compute IDs. You can use the compute ID to specify how the associated compute unit should behave differently from the other compute units that are derived from the same kernel source code. For example, you can use the return value of `get_compute_id()` to index into an array of channels to specify which channel each compute unit should read from or write to.

The `num_compute_units` attribute accepts up to three arguments (that is, `num_compute_units(X,Y,Z)`). In conjunction with the `get_compute_id()` function, this attribute allows you to create one-dimensional, two-dimensional, and three-dimensional logical arrays of compute units. An example use case of a 1D array of compute units is a linear pipeline of kernels (also called a daisy chain of kernels). An example use case of a 2D array of compute units is a systolic array of kernels.

Figure 23. Schematic Diagram of a 4x4 Array of Compute Units

The following example code specifies `num_compute_units(4,4)` in a single work-item kernel results in a 4x4 array that consists of 4 x 4 = 16 compute units.

```
__attribute__((max_global_work_dim(0)))
__attribute__((autorun))
__attribute__((num_compute_units(4,4)))
__kernel void PE() {

   row = get_compute_id(0);
   col = get_compute_id(1);

   …
}
```



For a 3D array of compute units, you can retrieve the X, Y, and Z coordinates of a compute unit in the logical compute unit array using `get_compute_id(0)`, `get_compute_id(1)`, and `get_compute_id(2)`, respectively. In this case, the API is very similar to the API of the work-item's intrinsic functions (that is, `get_global_id()`, `get_local_id()`, and `get_group_id()`).

Global IDs, local IDs, and group IDs can vary at runtime based on how the host invokes the kernel. However, compute IDs are known at compilation time, allowing the offline compiler to generate optimized hardware for each compute unit.

12.4.2. Using Channels with Kernel Copies

To implement channels within compute units (that is, replicated kernel copies), create an array of channels and then index into that array using the return value of `get_compute_id()`.

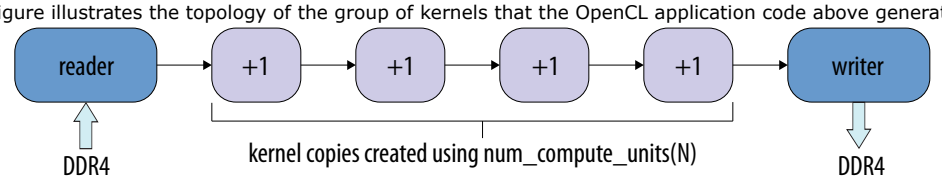
The example code below implements channels within multiple compute units.

```
#define N 4
channel int chain_channels[N+1];

__attribute__((max_global_work_dim(0)))
__kernel void reader(global int *data_in, int size) {
    for (int i = 0; i < size; ++i) {
```

```
        write_channel_intel(chain_channels[0], data_in[i]);
    }
}

__attribute__((max_global_work_dim(0)))
__attribute__((autorun))
__attribute__((num_compute_units(N)))
__kernel void plusOne() {
    int compute_id = get_compute_id(0);
    int input = read_channel_intel(chain_channels[compute_id]);
    write_channel_intel(chain_channels[compute_id+1], input + 1);
}

__attribute__((max_global_work_dim(0)))
__kernel void writer(global int *data_out, int size) {
    for (int i = 0; i < size; ++i) {
        data_out[i] = read_channel_intel(chain_channels[N]);;
    }
}
```

Figure 24. Example Topology of Kernel Copies that Implement Channels

This figure illustrates the topology of the group of kernels that the OpenCL application code above generates.



Note: The implementation of kernel copies is functionally equivalent to defining four separate kernels in your source code and then hard-coding unique indexes for the accesses to `chain_channels[N]`.

12.5. Intra-Kernel Registered Assignment Built-In Function

The Intel FPGA SDK for OpenCL Pro Edition provides the built-in function `__fpga_reg()` that you can include in your OpenCL kernel code. The `__fpga_reg()` function directs the Intel FPGA SDK for OpenCL Offline Compiler to insert at least one register between the operand and the return value of the function call.

In general, it is not necessary to include the `__fpga_reg()` function in your kernel code to achieve desired performance.

Attention: Intel strongly recommends that you use the `__fpga_reg()` function only if you are an experienced user of the Intel Quartus Prime Pro Edition software performing advanced optimization for a specific target device. You must have sufficient knowledge about the placement of portions of the data path on the FPGA.

Prototype of the `__fpga_reg()` built-in function:

```
T __fpga_reg(T op)
```

where *T* may be any sized type, such as standard OpenCL device data types, or a user-defined struct containing OpenCL types.

Use the `__fpga_reg()` function for the following purposes:

- Break the critical paths between spatially distant portions of a data path, such as between processing elements of a large systolic array.
- Reduce the pressure on placement and routing efforts caused by spatially distinct portions of the kernel implementation.

The `__fpga_reg()` function directs theIntel FPGA SDK for OpenCL Offline Compiler to insert at least one hardware pipelining register on the signal path that assigns the operand to the return value. This built-in function operates as an assignment in the OpenCL programming language, where the operand is assigned to the return value. The assignment has no implicit semantic or functional meaning beyond a standard C assignment. Functionally, you can think of the `__fpga_reg()` function being always optimized away by the offline compiler.

Note: The offline compiler does not provide feedback on where you should insert the `__fpga_reg()` function calls in your code. Use the Intel Quartus Prime Pro Edition software to determine where you should insert the calls to address specific aspects of performance.

You may introduce nested `__fpga_reg()` function calls in your kernel code to increase the minimum number of registers that the offline compiler inserts on the assignment path. Because each function call guarantees the insertion of at least one register stage, the number of calls provides a lower limit on the number of registers.

Consider the following example:

```
int out=__fpga_reg(__fpga_reg(in));
```

This line of code directs the offline compiler to insert at least two registers on the assignment path. The offline compiler may insert more than two registers on the path.

A. Support Statuses of OpenCL Features

The Intel FPGA SDK for OpenCL host runtime conforms with the OpenCL platform layer and application programming interface (API), with clarifications and exceptions.

A.1. Support Statuses of OpenCL 1.0 Features

The following sections outline the support statuses of the OpenCL features described in the *OpenCL Specification version 1.0*.

A.1.1. OpenCL1.0 C 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 Intel FPGA SDK for OpenCL 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. OpenCL programming language implementations that are supported with no additional clarifications are not shown.

Support Status column legend:

Symbol	Description
•	The feature is supported, and there might be a clarification for the supported feature in the Notes column
☐	The feature is supported with exceptions identified in the Notes column.
X	The feature is not supported.

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.

continued...

Section	Feature	Support Status	Notes
			Currently, the following double precision floating-point functions are expected to conform with the OpenCL Specification version 1.0: add / subtract / multiply / divide / ceil / floor / rint / trunc / fabs / fmax / fmin / sqrt / rsqrt / exp / exp2 / exp10 / log / log2 / log10 / sin / cos / asin / acos / sinh / cosh / tanh / asinh / acosh / atanh / pow / pown / powr / tanh / atan / atan2 / ldexp / log1p / sincos
	half precision float	☐	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. This feature is supported in the Emulator.
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 SDK 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 *Intel FPGA SDK for OpenCL Best Practices Guide* for tips on using `reqd_work_group_size` to improve kernel performance. The SDK 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 Intel FPGA SDK for OpenCL Offline Compiler 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 offline compiler to attempt to unroll the loop fully. Refer to the *Intel FPGA 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 SDK does not support images.
6.10	*Attribute Qualifiers*—The offline compiler parses attribute qualifiers as follows:		

continued...

Section	Feature	Support Status	Notes
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 offline compiler can parse attributes on various syntactic structures. It reserves some attribute names for its own internal use. Refer to the *Intel FPGA SDK for OpenCL Best Practices Guide* for tips on how to optimize kernel performance using these kernel attributes.
6.11.2	*Math Functions*		
	built-in math functions	☐	Preliminary support for built-in math functions for double precision float. These functions might not conform with the OpenCL Specification version 1.0.
	built-in `half_` and `native_` math functions	☐	Preliminary support for built-in `half_` and `native_` math functions for double precision float. These functions might not conform with the OpenCL Specification version 1.0.
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 SDK.
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.
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.

Related Information

- Intel FPGA SDK for OpenCL Pro Edition Best Practices Guide
- Argument Types for Built-in Geometric Functions on page 180

A.1.2. OpenCL C Programming Language Restrictions

The Intel FPGA SDK for OpenCL conforms with the OpenCL Specification restrictions on specific programming language features, as described in section 6.8 of the *OpenCL Specification version 1.0*.

Important: The Intel FPGA SDK for OpenCL Offline Compiler 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 offline compiler enforces the OpenCL restriction against pointer assignments between address spaces.
pointers to functions	X	The offline compiler 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 SDK does not support images.
bit fields	X	The offline compiler 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 offline compiler 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 offline compiler does not return an error for this restriction, but this feature is not supported.
irreducible control flow	X	The offline compiler does not return an error for this restriction, but this feature is not supported.
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 offline compiler does not enforce this restriction.
elements of a `struct` or a `union` belonging to different address spaces	X	The offline compiler does not enforce this restriction. ***Warning:*** Assigning elements of a `struct` or a `union` to different address spaces might cause a fatal error.

Support Status column legend:

Symbol	Description
•	The feature is supported, and there might be a clarification for the supported feature in the Notes column
☐	The feature is supported with exceptions identified in the Notes column.
X	The feature is not supported.

A.1.3. Argument Types for Built-in Geometric Functions

The Intel FPGA SDK for OpenCL 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		—

A.1.4. Numerical Compliance Implementation

Section 7 of the *OpenCL Specification version 1.0* describes features of the C99 and IEEE 754 standards that OpenCL-compliant devices must support. The Intel FPGA SDK for OpenCL 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	
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*	•	

A.1.5. Image Addressing and Filtering Implementation

The Intel FPGA SDK for OpenCL does not support image addressing and filtering. The SDK does not support images.

A.1.6. Atomic Functions

Section 9 of the *OpenCL Specification version 1.0* describes a list of optional features that some OpenCL implementations might support. The Intel FPGA SDK for OpenCL supports atomic functions conditionally. The implementation of the supported functions might not conform with the *OpenCL Specification Verison 1.0*.

- Section 9.5: *Atomic Functions for 32-bit Integers*—The SDK supports all 32-bit global and local memory atomic functions. The SDK 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 SDK 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.

A.1.7. Embedded Profile Implementation

Section 10 of the *OpenCL Specification version 1.0* describes the OpenCL embedded profile. The Intel FPGA SDK for OpenCL 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	•	64-bit integers are supported as a supplement to the *OpenCL Specification version 1.0* for the Embedded Profile.
2	3D images	X	The SDK does not support images.
3	Create 2D and 3D images with `image_channel_data_type` values	X	The SDK 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 SDK.
8	Error bounds listed for conversions from `CL_UNORM_INT8`, `CL_SNORM_INT8`, `CL_UNORM_INT16` and `CL_SNORM_INT16` to float	•	

A.2. Support Statuses of OpenCL 1.2 Features

The following sections outline the support statuses of the OpenCL features described in the *OpenCL Specification version 1.2*.

A.2.1. OpenCL 1.2 Runtime Implementation

The Intel FPGA SDK for OpenCL supports the implementation of sub-buffer objects and image objects. For more information on sub-buffer objects and image objects, refer to sections 5.2 and 5.3 of the *OpenCL Specification version 1.2*, respectively.

The SDK also supports the implementation of the following APIs:

- `clSetMemObjectDestructorCallback`
- `clGetKernelArgInfo`
- `clSetEventCallback`

For more information on these APIs, refer to sections 5.4.1, 5.7.3, and 5.9 of the *OpenCL Specification 1.2*, respectively.

Related Information

OpenCL Specification version 1.2

A.2.2. OpenCL 1.2 C Programming Language Implementation

The Intel FPGA SDK for OpenCL supports a number of OpenCL C programming language features that are specified section 6 of the *OpenCL Specification version 1.2*. The SDK conforms with the OpenCL C programming language with clarifications and exceptions.

Attention: The support status "•" means that the feature is supported, and there might be a clarification for the supported feature in the Notes column. The support status "□" means that the feature is supported with exceptions identified in the Notes column.

Table 15. Support Statuses of OpenCL 1.2 C Programming Language Features

Section	Feature	Support Status	Notes
6.1.3	Other Built-in Data Types	•	Preliminary support. This feature might not conform with the OpenCL Specification version 1.0.
6.12.12	*Miscellaneous Vector Functions*	•	The SDK supports implementations of the following additional built-in vector functions: • `vec_step` • `shuffle` • `shuffle2`
6.12.13	*printf*	□	Preliminary support. This feature might not conform with the OpenCL Specification version 1.0. See below for details.
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.			

continued...

Section	Feature	Support Status	Notes
During kernel execution, `printf` data is stored in a global `printf` buffer that the Intel FPGA SDK for OpenCL Offline Compiler 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. The format string for a `printf` statement cannot exceed 256 characters. 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

OpenCL Specification version 1.2

A.3. Support Statuses of OpenCL 2.0 Features

The following sections outline the support statuses of the OpenCL features described in the *OpenCL Specification version 2.0*.

A.3.1. OpenCL 2.0 Headers

The Intel FPGA SDK for OpenCL provides both the OpenCL 1.0 and OpenCL 2.0 headers by the Khronos Group.

Attention: The SDK currently does not support all OpenCL 2.0 APIs. If you use the OpenCL 2.0 headers and make a call to an unsupported API, the call will return with an error code to indicate that the API is not fully supported.

A.3.2. OpenCL 2.0 Runtime Implementation

The Intel FPGA SDK for OpenCL offers preliminary support for shared virtual memory implementation, as described in section 5.6 of the *OpenCL Specification version 2.0*. For more information on shared virtual memory, refer to section 5.6 of the *OpenCL Specification version 2.0*.

Important: Refer to your board's specifications to verify that your board supports shared virtual memory.

Related Information

OpenCL Specification version 2.0 (API)

A.3.3. OpenCL 2.0 C Programming Language Restrictions for Pipes

The Intel FPGA SDK for OpenCL offers preliminary support of OpenCL pipes.The following table lists the support statuses of pipe-specific OpenCL C programming language implementations, as described in the *OpenCL Specification version 2.0*

Attention: The support status "•" means that the feature is supported. There might be a clarification for the supported feature in the Notes column. A feature that is not supported by the SDK is identified with an "X".

Table 16. Support Statuses of Built-in Pipe Read and Write Functions

Details of the built-in pipe read and write functions are available in section 6.13.16.2 of the *OpenCL Specification version 2.0*.

Function	Support Status
`int read_pipe (pipe gentype p, gentype *ptr)`	•
`int write_pipe (pipe gentype p, const gentype *ptr)`	•
`int read_pipe (pipe gentype p, reserve_id_t reserve_id, uint index, gentype *ptr)`	X
`int write_pipe (pipe gentype p, reserve_id_t reserve_id, uint index, const gentype *ptr)`	X
`reserve_id_t reserve_read_pipe (pipe gentype p, uint num_packets)` `reserve_id_t reserve_write_pipe (pipe gentype p, uint num_packets)`	X
`void commit_read_pipe (pipe gentype p, reserve_id_t reserve_id)` `void commit_write_pipe (pipe gentype p, reserve_id_t reserve_id)`	X
`bool is_valid_reserve_id (reserve_id_t reserve_id)`	X

Table 17. Support Statuses of Built-in Work-Group Pipe Read and Write Functions

Details of the built-in pipe read and write functions are available in section 6.13.16.3 of the *OpenCL Specification version 2.0*.

Function	Support Status
`reserve_id_t work_group_reserve_read_pipe (pipe gentype p, uint num_packets)` `reserve_id_t work_group_reserve_write_pipe (pipe gentype p, uint num_packets)`	X
`void work_group_commit_read_pipe (pipe gentype p, reserve_id_t reserve_id)` `void work_group_commit_write_pipe (pipe gentype p, reserve_id_t reserve_id)`	X

Table 18. Support Statuses of Built-in Pipe Query Functions

Details of the built-in pipe query functions are available in section 6.13.16.4 of the *OpenCL Specification version 2.0*.

Function	Support Status
`uint get_pipe_num_packets (pipe gentype p)`	X
`uint get_pipe_max_packets (pipe gentype p)`	X

Related Information

OpenCL Specification version 2.0 (C Language)

A.4. Intel FPGA SDK for OpenCL Allocation Limits

Item	Limit
Maximum number of contexts	Limited only by host memory size
Maximum number of devices	128
Minimum global memory allocation by runtime	The runtime allocates 64 kB of device memory when the context is created. This memory is reserved for program variables in global address space and for static variables inside functions. If the OpenCL kernel uses the `printf` function, the runtime allocates an additional 64 kB of device memory.
Maximum number of queues	1024 ***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	Hardware: no static limit Emulator: 256
Maximum number of arguments per kernel	128
Maximum total size of kernel arguments	256 bytes per kernel
Maximum number of declared variables in the local memory per kernel	128

B. Document Revision History of the Intel FPGA SDK for OpenCL Pro Edition Programming Guide

Document Version	Intel Quartus Prime Version	Changes
2018.12.24	18.1.1	• Updated RTL Reset and Clock Signals on page 148 topic to discuss RTL module access of the system-wide clock that runs at twice the frequency of the OpenCL kernel clock. • Updated the description of the EXPECTED_LATENCY element in XML Elements for ATTRIBUTES on page 151 to include the EXPECTED_LATENCY value requirement for modules that can stall and require signals such as `iready`. • Updated the description of the IS_FIXED_LATENCY XML element in XML Elements for ATTRIBUTES on page 151 to include the EXPECTED_LATENCY value requirement when IS_FIXED_LATENCY="no" is set. • Added PARAMETER elements to XML Elements for ATTRIBUTES on page 151. • In the second bullet of #unique_199/unique_199_Connect_42_ul_rlt_5pj_w5 on page 158, removed "exactly one clock". • Revised Specifying the Name of an Intel FPGA SDK for OpenCL Offline Compiler Output File (-o <filename>) on page 107 to remove the = from the syntax of the `-o` option. The correct way to specify the name of the output file in the `aoc` command is `-o <filename>`. This option was documented incorrectly as `-o=<filename>`. Also, the syntax for this command option was corrected in examples throughout this publication.
2018.09.27	18.1	• Removed duplicate content in RTL Module Interfaces on page 146.

continued...

Document Version	Intel Quartus Prime Version	Changes
2018.09.24	18.1	• In Intel FPGA SDK for OpenCL Pro Edition, the Intel FPGA SDK for OpenCL Offline Compiler has a new front end. For a summary of changes introduced by this new front end, see *Improved Intel FPGA SDK for OpenCL Compiler Front End* in the Intel FPGA SDK for OpenCL Pro Edition Release Notes. • Added information about new OpenCL simulator preview in Debugging Your OpenCL Library Through Simulation (Preview) on page 161 and the following subtopics: — Compiling a Library for Simulation (-march=simulator) on page 162 — Simulating Your OpenCL Library on page 162 — Limitations of the Simulator on page 163 • Added information about the new OpenCL fast emulator preview in Using the Fast Emulator (Preview) on page 129 and the following subtopics: — Fast Emulator Environment Variables on page 131 — Extensions Supported by the Fast Emulator on page 132 — Fast Emulator Known Issues on page 132 • Revised Including Structure Data Types as Arguments in OpenCL Kernels on page 71 to indicate that `struct` kernel parameters must be passed either by value or as a pointer to `struct`. • Removed references to passing channels by reference because passing channels by reference is no longer supported.
2018.08.03	18.0	• Corrected a typo in Partitioning Buffers Across Different Memory Types (Heterogeneous Memory) on page 82. The correct flag `CL_MEM_HETEROGENEOUS_INTELFPGA`. Previously, the flag was incorrectly called `CL_MEM_HETEROGENEOUS_INTEL`.

continued...

Document Version	Intel Quartus Prime Version	Changes
2018.06.14	18.0	• Corrected the description of the behavior of the `aoc -c` command in Compiling Your Kernel without Building Hardware (-c) on page 105. With the `-c` command option, no folder or subdirectory is created.
2018.05.23	18.0	• Corrected code example in Using Channels with Kernel Copies on page 174 so that example compiles successfully.
2018.05.04	18.0	• Removed Intel Quartus Prime Standard Edition software-related information. • Updated `aoc` commands for intermediate compilation, and all relevant information, from `-c` to `-rtl`. Intel FPGA SDK for OpenCL Pro Edition provides the `-rtl` flag for intermediate compilation. • Increased the maximum number of devices to 128, as documented in the following topics: — Installing an FPGA Board (install) on page 18 — Querying the Device Name of Your FPGA Board (diagnose) on page 21 — Running a Board Diagnostic Test (diagnose <device_name>) on page 21 — Programming the FPGA Offline or without a Host (program <device_name>) on page 22 — Programming the Flash Memory (flash <device_name>) on page 22 — Programming Multiple FPGA Devices on page 97 • In One-Step Compilation for Simple Kernels on page 10, updated the figure *One-Step OpenCL Kernel Compilation Flow* and content of the topic. • In Multistep Intel FPGA SDK for OpenCL Pro Edition Design Flow on page 11, updated the figure *Multistep Intel FPGA SDK for OpenCL Pro Edition Design Flow* and inserted information on `-rtl`, fast compilation and incremental compilation. • Under Programming Strategies for Optimizing Data Processing Efficiency on page 24, added the topic Loop Concurrency (max_concurrency Pragma) on page 28. • Modified topic title *Programming Strategies for Optimizing Local Memory Efficiency* to Programming Strategies for Optimizing Pointer-to-Local Memory Size on page 31. • In Emulating I/O Channels on page 42, removed the section *Emulating Communication Between a Kernel and a Host or Other Process (Linux only)*. • Updated Emulating a Kernel that Passes Pipes or Channels by Value on page 122 to add information on passing pipes or channels by value, and that support for passing channels by reference is deprecated. • Modified the topic title Requirement for Multiple Command Queues in Channels or Pipes Implementation to Requirement for Multiple Command Queues to Execute Kernels Concurrently on page 80, and updated its content. • Added a Compiling Your Kernel Incrementally (-incremental) on page 115 topic with the following subtopics: — The Incremental Compile Report on page 116 — Additional Command Options for Incremental Compilation on page 118 — Limitations of the Incremental Compilation Feature on page 120 • Added a Compiling Your Kernel with Memory Error Correction Coding (-ecc) on page 120 topic for the ECC early feature. • In Direct Communication with Kernels via Host Pipes on page 64: — Retitled *New Optional Kernel Argument Attribute* to Optional intel_host_accessible Kernel Argument Attribute *on page 65* — Retitled *New API Functions* to API Functions for Interacting with cl_mem Pipe Objects Bound to Host-Accessible Pipe Kernel Arguments *on page 65*

Document Version	Intel Quartus Prime Version	Changes
		• In Intel FPGA SDK for OpenCL Allocation Limits on page 186: — Removed reference to the environment variable *CL_CONTEXT_PROGRAM_VARIABLES_TOTAL_SIZE_INTELFPGA* because the runtime no longer supports it. — Updated the maximum number of queues • In XML Syntax of an RTL Module on page 149, added information on the RESOURCES element. — Added the topic XML Elements for RESOURCES on page 154. • In Intel FPGA SDK for OpenCL Advanced Features on page 142, added the topic Intel Stratix® 10 Design-Specific Reset Requirements for Stall-Free and Stallable RTL Modules on page 149. • In Intel FPGA SDK for OpenCL Advanced Features on page 142, added the topic Intra-Kernel Registered Assignment Built-In Function on page 175

Date	Version	Changes
December 2017	2017.12.08	• Added the following new topics: — Profiling Autorun Kernels on page 137 — Profiling Enqueued and Autorun Kernels on page 86 — Profile Data Acquisition on page 87 — Multiple Autorun Profiling Calls on page 87 — Developing OpenCL Applications Using Intel Code Builder for OpenCL on page 138 — Configuring the Intel Code Builder for OpenCL Offline Compiler Plug-in for Microsoft Visual Studio on page 138 — Configuring the Intel Code Builder for OpenCL Offline Compiler Plug-in for Eclipse on page 138 — Creating a Session in the Intel Code Builder for OpenCL on page 139 — Configuring a Session on page 140 • In XML Syntax of an RTL Module on page 149, removed <PARAMETER name="WIDTH" value="32"/> from the XML specification file.
November 2017	2017.11.06	• Moved topics into separate chapters. • Rebranded references to the following: — The macro ALTERA_CL to INTELFPGA_CL. — The environment variable ALTERAOCLSDKROOT to INTELFPGAOCLSDKROOT. — The environment variable *CL_CONTEXT_PROGRAM_VARIABLES_TOTAL_SIZE_ALTERA*to *CL_CONTEXT_PROGRAM_VARIABLES_TOTAL_SIZE_INTELFPGA* — clGetExtensionFunctionAddress to clGetExtensionFunctionAddressIntelFPGA — The environment variable *CL_CONTEXT_EMULATOR_DEVICE_ALTERA* to *CL_CONTEXT_EMULATOR_DEVICE_INTELFPGA* — write_channel_altera to write_channel_intel — write_channel_nb_altera to write_channel_nb_intel — CL_MEM_BANK to CL_CHANNEL — CL_MEM_BANK_1_INTEL to CL_CHANNEL_1_INTELFPGA — CL_MEM_BANK_2_INTEL to CL_CHANNEL_2_INTELFPGA — Arria 10 to Intel Arria 10 — Quartus Prime to Intel Quartus Prime — Intel FPGA SDK for OpenCL Profiler to Intel FPGA Dynamic Profiler for OpenCL — TimeQuest Timing Analyzer to Timing Analyzer — Qsys Pro to Platform Designer • In Intel FPGA SDK for OpenCL FPGA Programming Flow on page 7, added FPGA data flow architecture diagram and related text. • In *Intel FPGA SDK for OpenCL Advanced Features* section, added RTL Module Interfaces on page 146 to provide example of how RTL module interfaces operate. • Updated the timing diagram in Avalon Streaming (Avalon-ST) Interface on page 147. • In Implementing Blocking Channel Writes on page 38 and Implementing Blocking Channel Reads on page 39, removed "which cannot be a constant" in the definition of <*type*>. • Added the topic Debugging Your OpenCL System That is Gradually Slowing Down on page 103.

continued...

Date	Version	Changes
		• Added a link to PLDA website in Compiling Your OpenCL Kernel on page 104. • Updated the last bullet point of Guidelines for Naming the Kernel on page 23 to include the keywords VHDL and Verilog. • In Intel FPGA SDK for OpenCL Advanced Features on page 142, listed the aspects of the design that can be controlled. • In OpenCL Library on page 142, added the expansion of RTL. • Split the sections of Understanding RTL Modules and the OpenCL Pipeline on page 143 into individual topics Overview: Intel FPGA SDK for OpenCL Pipeline Approach on page 144 and Integration of an RTL Module into the Intel FPGA SDK for OpenCL Pipeline on page 144. • In Overview: Intel FPGA SDK for OpenCL Pipeline Approach on page 144, aligned the left-hand example code with image on the right-hand. Moved the bottom portion of the image above the paragraph, which explains the image. • In Integration of an RTL Module into the Intel FPGA SDK for OpenCL Pipeline on page 144, added related links about Avalon-ST. • In Stall-Free RTL on page 145, split the paragraph into steps and added related links. • In Requirements for Deterministic Multiple Work-Item Ordering on page 34, added a third requirement for work-item ordering. • Updated Implementing Nonblocking Channel Reads on page 40. • Added new topic Speeding Up Your OpenCL Compilation (-fast-compile) on page 115 and implemented the convention `-option=<value>`. • In One-Step Compilation for Simple Kernels on page 10 and Multistep Intel FPGA SDK for OpenCL Pro Edition Design Flow on page 11, replaced references to .log file with HTML report and double dash command option with single dash. • In Compiling a Kernel for Emulation (-march=emulator) on page 123, added support for Stratix 10 • In Obtaining General Information on Software, Compiler, and Custom Platform on page 15, added a Notice to highlight that double dash and `-option <value>` conventions of aoc command are deprecated.

continued...

Date	Version	Changes
		• Implemented the conventions single dash and `-option=<value>` in the following topics: — Displaying the Compiler Version (-version) on page 15 — Listing the Intel FPGA SDK for OpenCL Offline Compiler Command Options (no argument, -help, or -h) on page 16 — Listing the Available FPGA Boards in Your Custom Platform (-list-boards) on page 17 — Partitioning Buffers Across Multiple Interfaces of the Same Memory Type on page 80 — Specifying the Location of Header Files (-I=<directory>) on page 106 — Specifying the Name of an Intel FPGA SDK for OpenCL Offline Compiler Output File (-o <filename>) on page 107 — Compiling a Kernel for a Specific FPGA Board (-board=<board_name>) on page 107 — Resolving Hardware Generation Fitting Errors during Kernel Compilation (-high-effort) on page 109 — Defining Preprocessor Macros to Specify Kernel Parameters (-D<macro_name>) on page 109 — Displaying the Estimated Resource Usage Summary On-Screen (-report) on page 112 — Disabling Burst-Interleaving of Global Memory (-no-interleaving=<global_memory_type>) on page 113 — Configuring Constant Memory Cache Size (-const-cache-bytes=<N>) on page 114 — Relaxing the Order of Floating-Point Operations (-fp-relaxed) on page 114 — Reducing Floating-Point Rounding Operations (-fpc) on page 114 — Emulating Channel Depth on page 123 — Compiling a Kernel for Emulation (-march=emulator) on page 123 — Packaging an OpenCL Helper Function File for an OpenCL Library on page 157 — OpenCL Library Command-Line Options on page 166 — Instrumenting the Kernel Pipeline with Performance Counters (-profile) on page 135 • In Accessing Custom Platform-Specific Functions on page 88, added related links to ICD loader. • In Specifying Number of Compute Units on page 30, Kernel Replication Using the num_compute_units(X,Y,Z) Attribute on page 172, and Emulating Your OpenCL Kernel on page 124, added a note on compute unit. • In Compiling a Kernel for Emulation (-march=emulator) on page 123, added support for Intel Stratix 10. • Added a new topic Discrepancies in Hardware and Emulator Results on page 127. • Added support status column legend OpenCL C Programming Language Restrictions on page 179 • Simplified the flowcharts in One-Step Compilation for Simple Kernels on page 10 and Multistep Intel FPGA SDK for OpenCL Pro Edition Design Flow on page 11 and updated the texts relevantly. • Removed references to *AOCL_BOARD_PACKAGE_ROOT* throughout the guide since it is deprecated. • Updated instances of `aocl install` to `aocl install <path_to_customplatform>`. • Updated instances of `aocl uninstall` to `aocl uninstall <path_to_customplatform>`

continued...

Date	Version	Changes
		• Added the following new topics for host pipes: — Direct Communication with Kernels via Host Pipes on page 64 — Optional intel_host_accessible Kernel Argument Attribute on page 65 — API Functions for Interacting with cl_mem Pipe Objects Bound to Host-Accessible Pipe Kernel Arguments on page 65 — Creating a Host Accessible Pipe on page 66 — Example Use of the cl_intel_fpga_host_pipe Extension on page 67 • In Enabling the Intel FPGA SDK for OpenCL Channels for OpenCL Kernel on page 37, added the pragma to enable the channel extension. • Updated the design example compilation procedures in Using an OpenCL Library that Works with Simple Functions (Example 1) on page 164 and Using an OpenCL Library that Works with External Memory (Example 2) on page 165. • In Restrictions in the Implementation of Intel FPGA SDK for OpenCL Channels Extension on page 35, replaced the Single Site call sub-section with Multiple Channel Call Site.
May 2017	2017.05.08	• Rebranded some functions in code examples as follows: — Rebranded `read_channel_altera` to `read_channel_intel`. — Rebranded `write_channel_altera` to `write_channel_intel`. — Rebranded `read_channel_nb_altera` to `read_channel_nb_intel`. — Rebranded `write_channel_nb_altera` to `write_channel_nb_intel`. — Rebranded `clGetBoardExtensionFunctionAddressAltera` to `clGetBoardExtensionFunctionAddressIntelFPGA`. • Added Emulating I/O Channels on page 42. • Added Implementing Arbitrary Precision Integers on page 68. • Added Coalescing Nested Loops on page 25. • Added Specifying a Loop Initiation interval (II) on page 27. • Added Emulating Channel Depth on page 123. • Added Avalon Streaming (Avalon-ST) Interface on page 147. • Removed all references to `#pragma OPENCL EXTENSION cl_altera_channels : enable` because this pragma is not required to implement channels. • Reorganized information related to heterogeneous memory as follows: — Merged *Specifying Pointer Size in Memory* content into Programming Strategies for Optimizing Pointer-to-Local Memory Size on page 31. — Restructured into three topics: • Allocating OpenCL Buffers for Manual Partitioning of Global Memory on page 80 • Partitioning Buffers Across Multiple Interfaces of the Same Memory Type on page 80 • Partitioning Buffers Across Different Memory Types (Heterogeneous Memory) on page 82 — Moved *Specifying Buffer Location in Global Memory* (previously under *Programming Strategies for Optimizing Access Efficiency*) content into Partitioning Buffers Across Different Memory Types (Heterogeneous Memory) on page 82. • Updated Collecting Profile Data During Kernel Execution on page 84 with a warning about the affect of collecting profile data on kernel launch times. • Updated Compiling Your OpenCL Kernel on page 104 with restrictions on compiling an encrypted `.cl` file. • Updated Restrictions and Limitations in RTL Support for the Intel FPGA SDK for OpenCL Library Feature on page 158 to indicate that an RTL module must use a single-input Avalon-ST interface to control inputs.

continued...

Date	Version	Changes
		• Updated topics affected by changes to the OpenCL profiler as follows: — Updated Launching the Intel FPGA Dynamic Profiler for OpenCL GUI (report) on page 136 with new command options. — Updated Figure 5 on page 12 in Multistep Intel FPGA SDK for OpenCL Pro Edition Design Flow on page 11 to reflect the new command options. • Corrected code example in Implementing Nonblocking Channel Reads on page 40. • Corrected code example in *Channel Execution in Loop with Multiple Work-Items* section of Work-Item Serial Execution of Channels on page 35. • In *Intel FPGA SDK for OpenCL Advanced Features* section, made the following updates: — Updated Interaction between RTL Module and External Memory on page 155 to indicate preferred method for RTL module and external memory interactions. — Updated Potential Incompatibility between RTL Modules and Partial Reconfiguration on page 156 to include link to the partial reconfiguration guidelines in the *Quartus Prime Pro Edition Handbook*. — Added information about `bankbits` and `merge`Allocating OpenCL Buffer for Manual Partitioning of kernel attributes to Kernel Attributes for Configuring Local and Private Memory Systems on page 167. — Rebranded some functions in code examples as follows: • Rebranded `read_channel_altera` to `read_channel_intel`. • Rebranded `write_channel_altera` to `write_channel_intel`.
October 2016	2016.10.31	• Rebranded the Altera SDK for OpenCL to Intel FPGA SDK for OpenCL. • Rebranded the Altera Offline Compiler to Intel FPGA SDK for OpenCL Offline Compiler. • Deprecated and removed support for big-endian system, resulting in the following documentation changes: — Removed the topic *Compiling a Kernel for a Big-Endian System (--big-endian)*. — Removed big-endian (64-bit) from the list of architectures that the host application can target. • Added the topic *Displaying the Compilation Environment of an OpenCL Binary* to introduce the `aoc env` command. • Removed *Adding Source References to Optimization Reports (-g)* because the offline compiler automatically includes source information in the compiler reports and enables symbolic debug during emulation on an x86 Linux machine. • Added the topic *Removing Debug Data from Compiler Reports and Source Code from the .aocx File (-g0)* to introduce the `-g0` `aoc` command option. • In *Limitations of the Intel FPGA SDK for OpenCL Emulator*, removed the limitation "The Emulator does not support half data type". • In *Linking Your Host Application to the Khronos ICD Loader Library*, provided an update that the Intel-supplied ICD Loader Library supports OpenCL Specification version 1.0 as well as implemented APIs from the OpenCL Specification versions 1.1, 1.2, and 2.0. • In *Managing an FPGA Board*, provided the following updates: — Noted that the SDK supports installation of multiple Custom Platforms. To use the SDK utilities on each board in a multi-board installation, the *AOCL_BOARD_PACKAGE_ROOT* environment variable setting must correspond to the Custom Platform subdirectory of the associated board. — Noted that in a system with multiple Custom Platforms, the host program should use ACD to discover the boards instead of directly linking to the MMD libraries.

continued...

Date	Version	Changes
		• Added the topic *Reviewing Your Kernel's report.html File* and included deprecation notice for the `analyze-area` utility option. As a result of introducing the HTML report, removed the following topics: — *Reviewing Your Kernel's Resource Usage Information in the Area Report* — *Accessing the Area Report* — *Layout of the Area Report* • In *Multistep Design Flow*, updated the design steps and the figure *The Multistep Intel FPGA SDK for OpenCL Design Flow* to replace area report with the HTML report, and remove information on enabling `-g`. • In *Inferring a Register*, corrected the text following the code snippet that explained how the offline compiler decide on the implementation of the array in hardware. • In *Linking to the ICD Loader Library* on Windows, updated the text to improve clarity. • In *Support Statuses of OpenCL Features* section, made the following updates: — Rebranded the Altera SDK for OpenCL to Intel FPGA SDK for OpenCL. — Rebranded the Altera Offline Compiler to Intel FPGA SDK for OpenCL Offline Compiler. — Modified information in the *Intel FPGA SDK for OpenCL Allocation Limits* section: • Updated information regarding minimum global memory allocation by runtime. • Updated the maximum number of queues from 70 to 256. • Updated the maximum number of kernels per FPGA device from 64 to no static limit when compiling to hardware and 256 when compiling to emulator. — In *OpenCL 1.0 C Programming Language Implementation*, under the Description column for half-precision float, added a note that this feature is supported in the Emulator. In addition, updated the support status of half-precision float from X to ☐. — Under *Support Statuses of OpenCL 2.0 Features*, added the topic *OpenCL 2.0 Headers* to explain that using the OpenCL 2.0 headers to call unsupported APIs will result in an error.
May 2016	2016.05.02	• Added a schematic diagram of the AOCL programming model in the *Altera SDK for OpenCL FPGA Programming Flow* section. • Moved the figure *The AOCL FPGA Programming Flow* to the *Altera Offline Compiler Kernel Compilation Flows* section. • Updated the figure *The Multistep AOCL Design Flow* and associated text to include the Review Area Report step. • Added information on the single-cycle floating-point accumulator feature for single work-item kernels. Refer to the *Single-Cycle Floating-Point Accumulator for Single Work-Item Kernels* section for more information. • Added information in the *Emulating Your OpenCL Kernel* section on multi-device support for emulation alongside other OpenCL SDKs using ICD.

continued...

Date	Version	Changes
		• Included information on the enhanced area report feature: — Added the option to invoke the `analyze-area` AOCL utility command to generate an HTML area report. — Included a topic that describes the layout of the HTML area report. • In *Linking to the ICD Loader Library on Windows*, removed `$(AOCL_LDLIBS)` from the code example for the modified `Makefile`. • In the *Multiple Work-Item Ordering* sections for channels and pipes, modified the characteristics that the AOCL uses to check whether the channel or pipe call is work-item invariant. • Added Intel FPGA SDK for OpenCL Advanced Feature section. • In *OpenCL 1.2 Runtime Implementation* under *Support Statuses of OpenCL Features* sections, noted that AOCL supports the `clSetEventCallback`, `clGetKernelArgInfo`, and `clSetMemObjectDestructorCallback` APIs.
November 2015	2015.11.02	• Added the option to invoke the `aoc` command with no argument to access the Altera Offline Compiler help menu. • Updated the *Multiple Host Threads* section to specify that the OpenCL host runtime is thread-safe. • Updated the following figure and sections to reflect multiple kernel source file support: — The figure *The AOCL FPGA Programming Flow* in the *AOCL FPGA Programming Flow* section — The *Compiling Your Kernel to Create Hardware Configuration File* section — The *Compiling Your Kernel without Building Hardware (-c)* section • In *Multiple Work-Item Ordering for Channels*, removed misleading text. • Updated the *Overview of Channels Implementation* figure. • Updated the following sections on OpenCL pipes: — *Overview of a Pipe Network Implementation* figure in *Overview of the OpenCL Pipe Functions* — Emulation support in *Restrictions in OpenCL Pipes Implementation* section — Replaced erroneous code with the correct syntax — Added link to *Implementing I/O Pipes Using the io Attribute* in *Declaring the Pipe Handle* • Added a reminder in *Programming an FPGA via the Host* that you should release an event object after use to prevent excessive memory usage. • In *Support Statuses of OpenCL Features* section, made the following updates: — Categorized feature support statuses and limitations based on OpenCL Specification versions. — Added the following functions to the list of OpenCL-conformant double precision floating-point functions: sinh / cosh / tanh / asinh / acosh / atanh / pow / pown / powr / tanh / atan / atan2 / ldexp / log1p / sincos — In *OpenCL 1.2 Runtime Implementation*, added sub-buffer object support. — In *OpenCL 2.0 Runtime Implementation*, added preliminary shared virtual memory support. — In *Altera SDK for OpenCL Allocation Limits*, added a minimum global memory allocation limit by the runtime.

continued...

Date	Version	Changes
May 2015	15.0.0	• In *Guidelines for Naming the Kernel*, added entry that advised against naming an OpenCL kernel `kernel.cl`. • In *Instrumenting the Kernel Pipeline with Performance Counters (--profile)*, specified that you should run the host application from a local disk to avoid potential delays caused by slow network disk accesses. • In *Emulating and Debugging Your OpenCL Kernel*, modified Caution note to indicate that you must emulate a design targeting an SoC on a non-SoC board. • In *Emulating Your OpenCL Kernel*, updated command to run the host application and added instruction for overriding default temporary directory containing `<process_ID>-libkernel.so`. • Introduced the `--high-effort` aoc command flag in *Resolving Hardware Generation Fitting Errors during Kernel Compilation*. • In *Enabling Double Precision Floating-Point Operations*, introduced the `OPENCL EXTENSION` pragma for enabling double precision floating-point operations. • Introduced OpenCL pipes support. Refer to *Implementing OpenCL Pipes* (and subsequent subtopics) and *Creating a Pipe Object in Your Host Application* for more information. • In *AOCL Channels Extension: Restrictions*, added code examples to demonstrate how to statically index into arrays of channel IDs. • In *Multiple Host Threads*, added recommendation for synchronizing OpenCL host function calls in a multi-threaded host application. • Introduced ICD and ACD support. Refer to *Linking Your Host Application to the Khronos ICD Loader Library* for more information. • Introduced `clGetBoardExtensionFunctionAddressAltera` for referencing user-accessible functions. Refer to *Accessing Custom Platform-Specific Functions* for more information. • In *Support Statuses of OpenCL Features* section, made the following updates: — Listed the double precision floating-point functions that the Altera® SDK for OpenCL supports preliminarily. — Added *OpenCL C Programming Language Restrictions for Pipes*.
December 2014	14.1.0	• Reorganized information flow. Information is now presented based on the tasks you might perform using the Altera SDK for OpenCL (AOCL) or the Altera RTE for OpenCL. • Removed information pertaining to the `--util` <N> and `-O3` Altera Offline Compiler (AOC) options. • Added the following information on PLDA QuickUDP IP core licensing in *Compiling Your OpenCL Kernel*: 1. A PLDA QuickUDP IP core license is required for the Stratix V Network Reference Platform or a Custom Platform that uses the QuickUDP IP core. 2. Improper installation of the QuickUDP IP core license causes compilation to fail with an error message that refers to the QuickTCP IP core. • Added reminder that conditionally shifting a large shift register is not recommended. • Removed the *Emulating Systems with Multiple Devices* section. A new env `CL_CONTEXT_EMULATOR_DEVICE_ALTERA=<number_of_devices>` command is now available for emulating multiple devices. • Removed language support limitation from the *Limitations of the AOCL Emulator* section. • In *AOCL Allocation Limits* under *Support Statuses of OpenCL Features* section, updated the maximum number of kernels per FPGA device from 32 to 64.

continued...

Date	Version	Changes
June 2014	14.0.0	• Removed the `--estimate-throughput` and `--sw-dimm-partition` AOC options • Added the `-march=emulator`, `-g`, `--big-endian`, and `--profile` AOC options • `--no-interleaving` needs <*global_memory_type*> argument • `-fp-relaxed=true` is now `--fp-relaxed` • `-fpc=true` is now `--fpc` • For non-SoC devices, `aocl diagnostic` is now `aocl diagnose` and `aocl diagnose` <*device_name*> • `program` and `flash` need <*device_name*> arguments • Added *Identifying the Device Name of Your FPGA Board* • Added *AOCL Profiler Utility* • Added *AOCL Channels Extension* and associated subsections • Added *Attributes for Channels* • Added *Match Data Layouts of Host and Kernel Structure Data Types* • Added *Register Inference* and *Shift Register Inference* • Added *Channels and Multiple Command Queues* • Added *Shared Memory Accesses for OpenCL Kernels Running on SoCs* • Added *Collecting Profile Data During Kernel Execution* • Added *Emulate and Debug Your OpenCL Kernel* and associated subsections • Updated *AOC Kernel Compilation Flows* • Updated *-v* • Updated *Host Binary Requirement* • Combined *Partitioning Global Memory Accesses* and *Partitioning Heterogeneous Global Memory Accesses* into the section *Partitioning Global Memory Accesses* • Updated *AOC Allocation Limits* in *Appendix A* • Removed `max_unroll_loops`, `max_share_resources`, `num_share_resources`, and `task` kernel attributes • Added `packed`, and `aligned(`<*N*>`)` kernel attributes • In *Support Statuses of OpenCL Features* section, updated the following AOCL allocation limits: — Maximum number of contexts — Maximum number of queues — Maximum number of even objects per context
December 2013	13.1.1	• 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. — section. — Added the subsection *Programming Multiple FPGA Devices* under *FPGA Programming*.

continued...

Date	Version	Changes
		• 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*.
November 2013	13.1.0	• Reorganized information flow. • Updated and renamed *Intel FPGA 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*.
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*.

continued...

Date	Version	Changes
		• 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 clCreateProgramWithBinary* 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.
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.