Altera SDK for OpenCL

Programming Guide

Subscribe
Send Feedback

OCL002-13.1.1
2013.12.13

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

Contents

Altera SDK for OpenCL Programming Guide

1

2013.12.13

OCL002-13.1.1 Subscribe Send Feedback

About Altera SDK for OpenCL Programming Guide

The *Altera Software Development Kit* (SDK) *for OpenCL Programming Guide* describes the contents and functionality of the Altera SDK for OpenCL (AOCL) version 13.1. The AOCL[1] is an OpenCL[2]-based heterogeneous parallel programming environment for Altera field programmable gate arrays (FPGAs).

This document assumes that you have read the *Altera SDK for OpenCL Getting Started Guide*, and have performed the following tasks:

- Download and install the Quartus® II software version 13.1.
- Download and install the Stratix® V device support.
- Download and install the AOCL version 13.1.
- Install your FPGA board.
- Program your FPGA with the hello_world example OpenCL application.

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

Audience

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

Related Information

- **Altera SDK for OpenCL Gettting Started Guide**
  Refer to the AOCL Getting Started Guide for installation instructions for the AOCL.
- **OpenCL Specification version 1.0**
  Refer to the *OpenCL Specification version 1.0* for detailed information on the OpenCL API and programming language.

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

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

Contents of the AOCL Version 13.1

The AOCL version 13.1 provides logic components, drivers, and AOCL-specific libraries and files.

Logic Components

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

  The host runtime consists of the following libraries:

  - Statically-linked libraries provide OpenCL host APIs, hardware abstractions and helper libraries.
  - Dynamically-linked libraries (DLLs) provide hardware abstractions and helper libraries.

Drivers, Libraries and Files

On Windows and Linux machines, the AOCL software installation process installs the AOCL into a folder or directory referenced by the *ALTERAOCLSDKROOT* environment variable. The table below highlights some of the contents of the AOCL version 13.1.

Table 1-1: Contents of the AOCL for Windows and Linux

Windows Folder	Linux Directory	Description
\bin	**/bin**	User commands in the AOCL. Include this folder or directory in your *PATH* environment variable.
\windows64\driver	**/linux64/driver**	The board driver in this folder or directory allows your host computer to communicate with the FPGA board.
\board	**/board**	The Altera Preferred Board Partner Program (APBPP) for OpenCL board package for each FPGA board supported by the AOCL.
\ip	**/ip**	Intellectual property (IP) core used to compile device kernels.
\host	**/host**	Files necessary for compiling your host program.
\host\include	**/host/include**	OpenCL version 1.0 header files and AOCL interface files necessary for compiling and linking your host program. Add this path to the **include** file search path in your development environment.
\host\windows64\lib	**/host/linux64/lib**	OpenCL host runtime libraries that provide the OpenCL platform and runtime APIs. These libraries are necessary for linking your host program. To run an OpenCL application on Linux, include this directory in the *LD_LIBRARY_PATH* environment variable.

Windows Folder	Linux Directory	Description
\host\windows64\bin		DLLs necessary for running your host program. Include this folder in your *PATH* environment variable.

Example OpenCL Applications

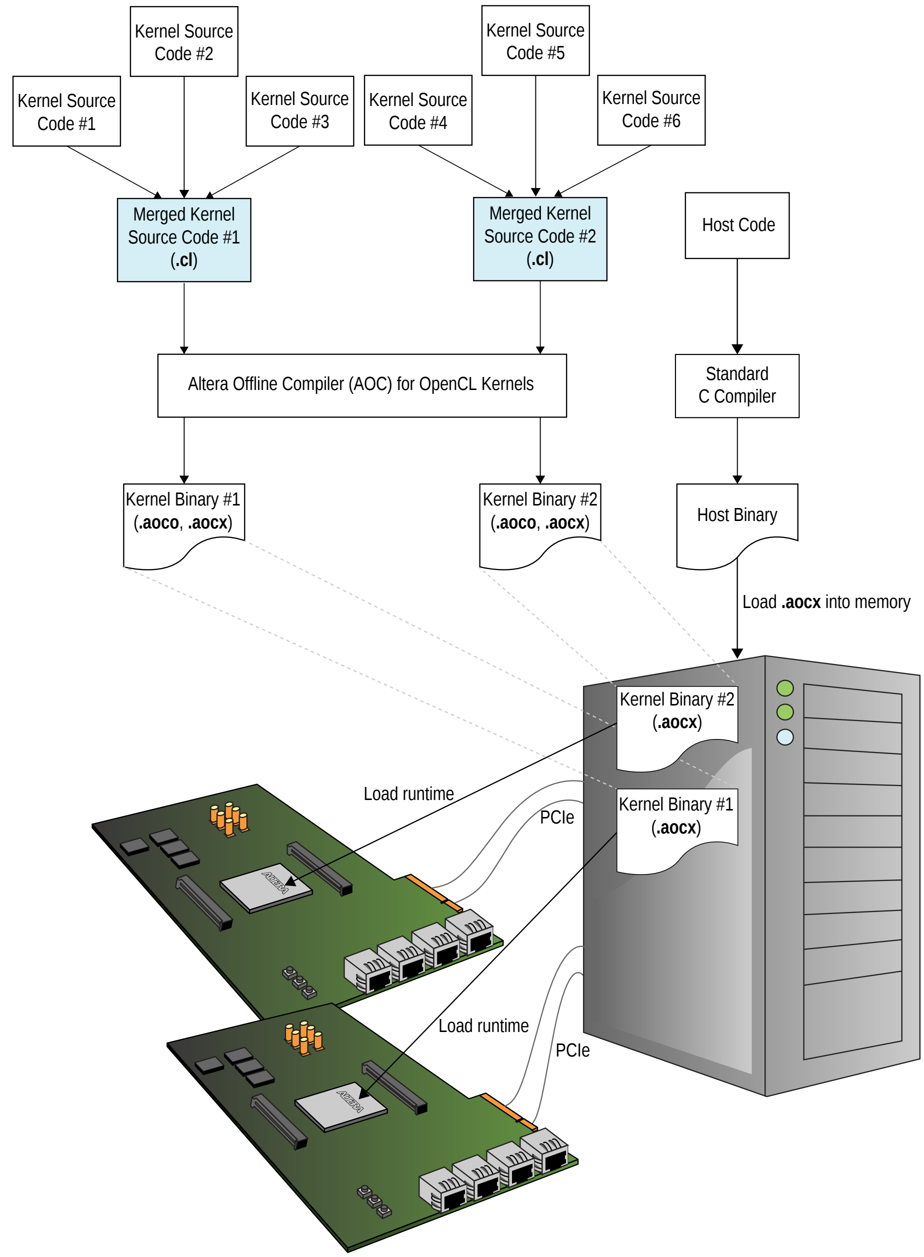
You can download example OpenCL applications from the **OpenCL Design Examples** page on the Altera website.

AOCL FPGA Programming Flow

The AOCL programs an FPGA with an OpenCL application in a two-step process. The AOC first compiles your OpenCL kernels, and then the host-side C compiler compiles your host application and links the OpenCL kernels to it.

The following figure depicts the AOCL FPGA programming flow:

Figure 1-1: The AOCL FPGA Programming Flow



Important: Before you compile your OpenCL kernels, you must consolidate your kernel source files into a single **.cl** source file.

The OpenCL kernel source file (**.cl**) contains your OpenCL source code. The AOC compiles your kernel and generates the following files and folders:

- The *Altera Offline Compiler Object file* (**.aoco**), which contains kernel and configuration information necessary at runtime.
- The *Altera Offline Compiler Executable file* (**.aocx**), which is the hardware configuration file.
- The <*your_kernel_filename*> folder or subdirectory, which contains data necessary to create the **.aocx** file.

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

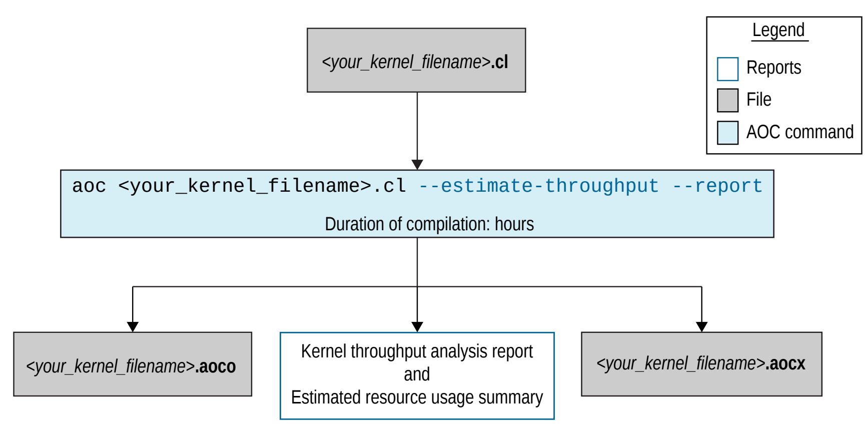
AOC Kernel Compilation Flows

The AOC can create your FPGA hardware configuration file in a one-step or a two-step process. The complexity of your kernel dictates the AOC compilation option you implement.

One-Step Compilation for Simple Kernels

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

Figure 1-2: One-Step AOC Compilation Flow



If you do not require the AOC to perform an estimated kernel throughput analysis, type the command `aoc <your_kernel_filename>.cl` to generate the hardware configuration file in a single step. During compilation, the AOC generates both the **.aoco** and the **.aocx** files.

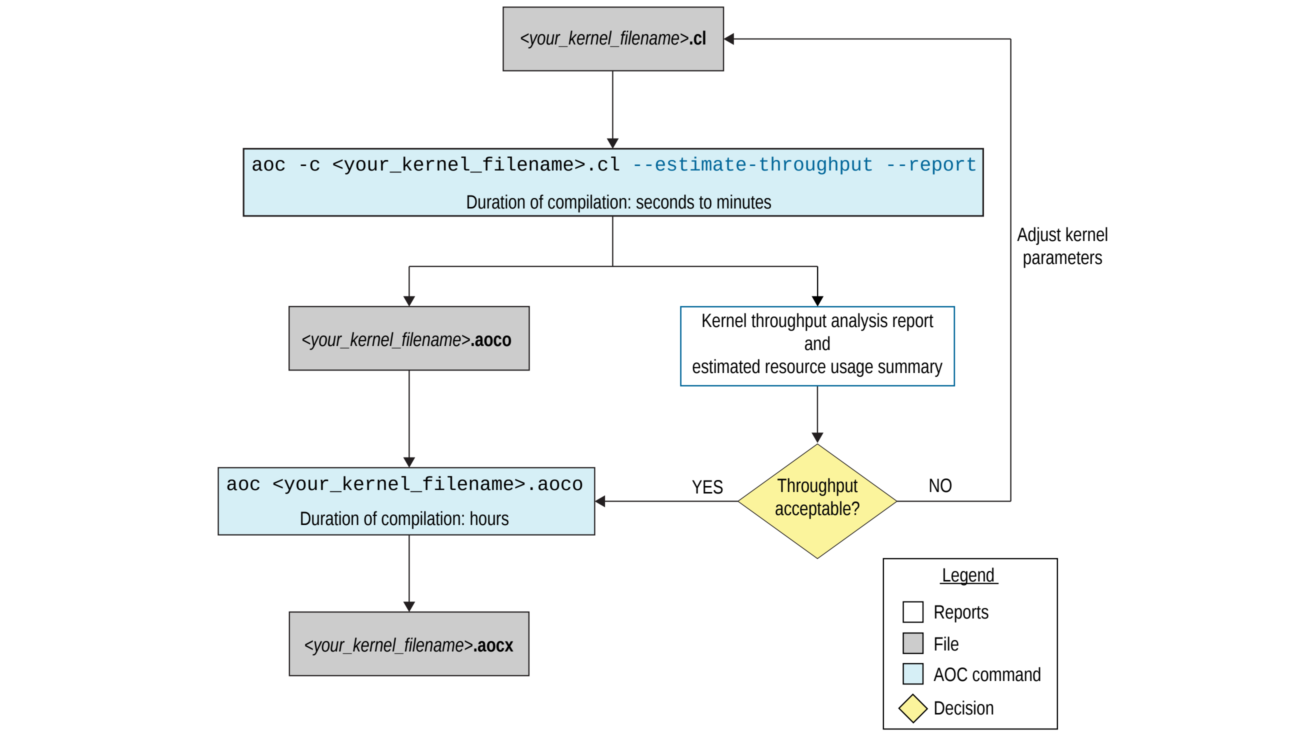
Important: The process of creating the **.aocx** file takes hours to complete.

Two-Step Compilation for Complex Kernels

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

The figure below illustrates the two-step compilation flow:

Figure 1-3: Two-Step AOC Compilation Flow



To perform the first stage of compilation, include the `-c` option in the `aoc` command. The AOC generates the **.aoco** file and a *<your_kernel_filename>* folder or subdirectory. This compilation step only takes seconds to minutes to complete. After you finalize your kernel code, you can perform the second step of the compilation process to create the hardware configuration file.

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

Compilation Reports

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

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

When you adjust your kernel code, you can direct the AOC to calculate throughput estimates, based on a set of heuristics, by including the `--estimate-throughput` option in your `aoc` command. The AOC generates an estimated throughput analysis report in the *<your_kernel_filename>*.**log** file. In addition, the AOC calculates the estimated resource usage by default during compilation. You can review a summary of the estimated resource usage in the **.log** file.

Alternatively, if you want to display both the kernel throughput analysis and the estimated resource usage summary on-screen, you can invoke the following command:

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

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

Related Information

- -c on page 1-11
- -v on page 1-8
- --report on page 1-8
- --estimate-throughput on page 1-9

AOC Options

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

AOC Help Options on page 1-7
You may include AOC help options in your `aoc` command to obtain information on the software, and on the compilation process.

AOC Overall Options on page 1-10
The AOC includes command line options that allow you to customize the compilation process such as compiling OpenCL kernels without hardware build, specifying names of output files, and defining macros.

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

AOC Optimization Controls on page 1-14
The AOC optimization controls direct the AOC to perform tasks such as partitioning memory, specifying logic utilization threshold, and modifying floating-point operations.

Related Information

Vector Addition OpenCL Design Example
You may download the vectorAdd OpenCL design example and compare the outputs of your `aoc` command invocations with the documented outputs.

AOC Help Options

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

--version on page 1-8
To direct the AOC to print out the AOCL version and then exits, invoke the `aoc --version` command.

--help or -h on page 1-8
To display a list of `aoc` command options, invoke the `aoc --help` or `aoc -h` command.

-v on page 1-8
To direct the AOC to report on the progress of a full compilation, invoke the `aoc -v <your_kernel_filename>.cl` command. To direct the AOC to report on the progress of a compilation that does not include the hardware build, invoke the `aoc -c -v <your_kernel_filename>.cl` command.

--report on page 1-8
If you want to review the estimated resource usage summary on-screen, invoke the `aoc <your_kernel_filename>.cl --report` command.

--estimate-throughput on page 1-9
To direct the AOC to calculate the estimated kernel throughput, invoke the `aoc <your_kernel_filename>.cl --estimate-throughput` command.

--version

To direct the AOC to print out the AOCL version and then exits, invoke the `aoc --version` command.

Example output:

```
Altera SDK for OpenCL, 64-Bit Offline Compiler
Version 13.1 Build 162
Copyright (C) 2013 Altera Corporation
```

--help or -h

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

-v

To direct the AOC to report on the progress of a full compilation, invoke the `aoc -v <your_kernel_filename>.cl` command. To direct the AOC to report on the progress of a compilation that does not include the hardware build, invoke the `aoc -c -v <your_kernel_filename>.cl` command.

For example, when you invoke the command `aoc -c -v vectorAdd.cl`, the AOC informs you of the compilation step it is performing, as shown below:

```
aoc: Environment checks are completed successfully.
aoc: Selected target board pcie385n_a7
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 vectorAdd.aoco"
```

--report

By default, the AOC estimates hardware resource usage during compilation. The AOC factors in the usage of external interfaces such as PCIe, memory controller, and direct memory access (DMA) engine in its calculations. You can review the estimated resource usage summary in the *<your_kernel_filename>*.**log** file located in the *<your_kernel_filename>* folder or subdirectory. If you want to review the estimated resource usage summary on-screen, invoke the `aoc <your_kernel_filename>.cl --report` command.

For example, the AOC generates the following compilation report for the vectorAdd example OpenCL application when you invoke the `aoc -c vectorAdd.cl --report` command:

```
aoc: Selected target board pcie385n_a7

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

--estimate-throughput

To direct the AOC to calculate the estimated kernel throughput, invoke the `aoc <your_kernel_filename>.cl --estimate-throughput` command.

Important: The data reported in the kernel throughput analysis are estimates derived from a set of heuristics. The information does not represent actual kernel performance values. Use the values reported in the kernel throughput analysis report as references for visualizing how kernel performance changes when you adjust your kernel code.

To direct the AOC to report on the progress of a compilation that does not perform a hardware build, invoke the `aoc -c <your_kernel_filename>.cl --estimate-throughput` command.

Tip: If you want to review the kernel throughput analysis and the estimated resource usage summary on-screen, invoke the following command:

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

You can review the estimated kernel throughput analysis by accessing the <*your_kernel_filename*>**.log** file, located in the <*your_kernel_filename*> folder or subdirectory.

For example, the **.log** file contains the following compilation reports for the vectorAdd example OpenCL application when you invoke the `aoc -c vectorAdd.cl --estimate-throughput` command:

```
Kernel throughput analysis for : vectorAdd
.. simd work items : 1
.. compute units : 1
.. throughput due to control flow analysis : 240.00 M work items/second
.. kernel global memory bandwidth analysis : 3031.58 MB/second
.. kernel number of local memory banks : none
+--------------------------------------------------------------------+
; Estimated Resource Usage Summary                                   ;
+----------------------------------------+---------------------------+
; Resource                               + Usage                     ;
+----------------------------------------+---------------------------+
; Logic utilization                      ; 18%                       ;
; Dedicated logic registers              ; 7%                        ;
; Memory blocks                          ; 16%                       ;
```

```
; DSP blocks                             ; 0%                          ;
+----------------------------------------+----------------------------;
System name: vectorAdd
```

- **simd work items**—reports the number of work-items executed in the single instruction multiple data (SIMD) programming model through the compute unit.

  The AOC has achieved the requested SIMD configuration if the value in the report matches the `num_simd_work_items` attribute value you specify in the OpenCL kernel code.

- **compute units**—reports the number of compute units the AOC creates for your kernel.
- **throughput due to control flow analysis**—provides an estimate of the number of work-items that the hardware implementation of your kernel can process per second, excluding global memory bandwidth limitations and local memory stalling.

  Attention: If your kernel contains a loop with an unknown trip count, the AOC assumes a trip count of 1024 iterations for static analysis purposes. As a result, the estimated value for **throughput due to control flow analysis** might not reflect accurately the actual work-items per second throughput that your kernel achieves.

- **kernel global memory bandwidth analysis**—reports the peak global memory bandwidth necessary to achieve the throughput specified by the **throughput due to control flow analysis**.
- **kernel number of local memory banks**—reports the number of local memory banks implemented to achieve estimated throughput.

Depending on the complexity of your kernel, the AOC might analyze additional throughput parameters, as shown below:

- **total # of RAMs used (local mem)/compute unit**—reports the number of on-chip RAM blocks each compute unit uses to implement local memory in the FPGA.
- **kernel number of local memory masters**—reports the total number of local memory accesses by load and store operations.
- **kernel local avg port fanin**—provides a measure of the complexity of local memory interconnects.
- **reducing fmax because of connectivity**—provides an estimate of the kernel's F_{max} given the complexity of the memory interconnect between compute units and local memory.
- **local stalls derate throughput by**—reports, as a number between 0.00 and 1.00, the degree of kernel stalling due to local memory contention.

  A value of 1.00 indicates no memory contention.

- **because of local stalls throughput**—provides an estimate of the number of work-items that the hardware implementation of your kernel can process per second, after accounting for global memory bandwidth limitations and local memory stalling.

AOC Overall Options

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

-c on page 1-11
To direct the AOC to compile your OpenCL kernel and generate a Quartus II hardware design project without creating a hardware configuration file, invoke the `aoc -c <your_kernel_filename>.cl` command.

-o <filename> on page 1-11
To assign a specific filename to the output file, include the `-o <filename>` flag in your `aoc` command.

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

-D <macro_name> or -D <macro_name=value> on page 1-12
You can define a preprocessor macro in your kernel source file by including the `-D <macro_name>` or `-D <macro_name=value>` flag in your `aoc` command.

-W on page 1-12
If you want to suppress all warning messages, include the `-W` flag when you invoke the `aoc` command.

-Werror on page 1-12
If you want to convert all warning messages into error messages, include the `-Werror` flag in your `aoc` command.

-c

To direct the AOC to compile your OpenCL kernel and generate a Quartus II hardware design project without creating a hardware configuration file, invoke the `aoc -c <your_kernel_filename>.cl` command.

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

-o *<filename>*

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

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

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

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

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

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

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

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

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

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

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

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

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

-I *<directory>*

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

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

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

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

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

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

or

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

Related Information

Preprocessor Macros on page 1-20

-W

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

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

-Werror

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

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

AOC Modifiers

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

--list-boards on page 1-13
To print out a list of FPGA boards available in your board package, include the `--list-boards` flag in the `aoc` command.

--board *<board_name>* on page 1-13
To compile your OpenCL kernel for a specific FPGA board, invoke the `aoc --board <board_name> <your_kernel_filename>.cl` command.

--list-boards

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

If you install a board package that includes more than one board type, when you invoke the `aoc --list-boards` command, the AOC generates an output that resembles the following:

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

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

Remember: To view the list of available boards, you must first set the environment variable *AOCL_BOARD_PACKAGE_ROOT* to point to the location of the board package.

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

--board *<board_name>*

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

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

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

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

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

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

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

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

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

and then

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

If you have an accelerator board consisting of two FPGAs, each FPGA device has an equivalent "board" name. If you have two FPGAs on a single accelerator board (for example, `board_fpga1` and `board_fpga2`), and you want to target a **kernel_1.cl** to `board_fpga1` and a **kernel_2.cl** to `board_fpga2`, invoke the following commands:

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

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

AOC Optimization Controls

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

--sw-dimm-partition on page 1-14
You can include the `--sw-dimm-partition` flag in the `aoc` command to manage buffer placement in global memory manually.

--no-interleaving on page 1-15
You can disable burst-interleaving for all global memory banks of the same type and manage them manually by including the `--no-interleaving <memory_type>` option in your `aoc` command.

--const-cache-bytes *<N>* on page 1-15
Include the `--const-cache-bytes <N>` flag in your `aoc` command to direct the AOC to configure the constant memory cache size (rounded up to the closest power of 2).

-O3 on page 1-16
To apply resource-driven optimizations to improve performance without violating fitting requirements, invoke the `aoc -O3 <your_kernel_filename>.cl` command.

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

-fp-relaxed=true on page 1-16
The `-fp-relaxed=true` flag directs the AOC to relax the order of arithmetic floating-point operations using a balanced tree hardware implementation.

-fpc=true on page 1-16
The `-fpc=true` flag directs the AOC to remove intermediary floating-point rounding operations and conversions whenever possible, and to carry additional bits to maintain precision.

--sw-dimm-partition

You can include the `--sw-dimm-partition` flag in the `aoc` command to manage buffer placement in global memory manually. Manual partitioning of memory buffers overrides the default burst-interleaved configuration of global memory.

To direct the AOC to partition the global memory into the two banks manually, invoke the `aoc --sw-dimm-partition <your_kernel_filename>.cl` command.

After you configure the global memory, allocate each buffer in one bank or the other with the Altera-specific `cl_mem_flags` (for example, `CL_MEM_BANK_2_ALTERA`).

Related Information

Partitioning Global Memory Accesses on page 1-32

--no-interleaving

The AOC cannot burst-interleave global memory across different memory types. You can disable burst-interleaving for all global memory banks of the same type and manage them manually by including the `--no-interleaving <memory_type>` option in your `aoc` command.

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

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

Note: Disabling burst-interleaving for all the heterogeneous memory banks on your FPGA board is similar to including the `--sw-dimm-partition` option in your `aoc` command.

If you only include the `--no-interleaving <memory_type>` option for one of the memory types (for example, `aoc <your_kernel_filename>.cl --no-interleaving DDR`), the AOC configures the QDR memory bank in a burst-interleaved fashion, and enables manual partitioning for the DDR memory bank.

Related Information

Partitioning Heterogeneous Global Memory Accesses on page 1-33

--const-cache-bytes <N>

Include the `--const-cache-bytes <N>` flag in your `aoc` command to direct the AOC to configure the constant memory cache size (rounded up to the closest power of 2).

The default constant cache size is 16 kilobytes (kB). To configure the constant memory cache size, invoke the following command:

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

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

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

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

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

Related Information

Altera SDK for OpenCL Optimization Guide

For more information on optimizing constant memory accesses, refer to the *Constant Cache Memory* section of the AOCL Optimization Guide.

-O3

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

The estimated logic utilization should be less than or equal to 85%.

Related Information

Altera SDK for OpenCL Optimization Guide

For more information, refer to the *Resource-Driven Optimization* section of the AOCL Optimization Guide.

--util <*N*>

By default, the AOC performs resource-driven optimizations assuming that the logic utilization threshold is 85%. Include the `--util <N>` flag in your `aoc -O3 <your_kernel_filename>.cl` command to override the default logic utilization threshold.

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

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

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

Related Information

Altera SDK for OpenCL Optimization Guide

For more information, refer to the *Resource-Driven Optimization* section of the AOCL Optimization Guide.

-fp-relaxed=true

The `-fp-relaxed=true` flag directs the AOC to relax the order of arithmetic floating-point operations using a balanced tree hardware implementation.

Caution: To implement this optimization control, your program must be able to tolerate small variations in the floating-point results.

To direct the AOC to execute a balanced tree hardware implementation, invoke the following command:

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

Related Information

Altera SDK for OpenCL Optimization Guide

For more information, refer to the *Floating-Point Operations* section of the AOCL Optimization Guide.

-fpc=true

The `-fpc=true` flag directs the AOC to remove intermediary floating-point rounding operations and conversions whenever possible, and to carry additional bits to maintain precision. Implementing this optimization control also changes the rounding mode to round towards zero only at the end of a chain of floating-point arithmetic operations (that is multiplications, additions, and subtractions).

To direct the AOC to reduce the number of rounding operations, invoke the following command:

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

Related Information

Altera SDK for OpenCL Optimization Guide

For more information, refer to the *Floating-Point Operations* section of the AOCL Optimization Guide.

AOCL Utility

The AOCL utility provides you with additional functionality to perform high-level tasks. You can obtain information on your host program, such as flags and makefile fragments.

General AOCL Utilities on page 1-17
The following AOCL utility options perform general tasks such as providing version and help information.

AOCL Utilities for Building Your Host Program on page 1-17
The following AOCL utility options provide components such as flags and libraries for compiling and linking your host program.

AOCL Utilities for Managing an FPGA Board on page 1-19
The following AOCL utility options direct the AOCL to perform tasks such as installing and programming your FPGA board, and running diagnostic tests.

General AOCL Utilities

The following AOCL utility options perform general tasks such as providing version and help information.

version

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

Example output:

```
Altera SDK for OpenCL, 64-Bit Offline Compiler
Version 13.1 Build 162
Copyright (C) 2013 Altera Corporation
```

help and help *<subcommand>*

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

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

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

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

Usage: aocl install

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

AOCL Utilities for Building Your Host Program

The following AOCL utility options provide components such as flags and libraries for compiling and linking your host program.

example-makefile or makefile

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

compile-config

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

ldflags

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

Attention: This command does not list the libraries themselves.

ldlibs

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

link-config or linkflags

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

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

Compiling and Linking Your Host Program

In an OpenCL application, the host program uses standard OpenCL runtime application programming interfaces (APIs) to manage device configuration, data buffers, kernel launches, and even synchronization. The host program also contains host-side functions such as file I/O, or portions of the source code that do not run on an accelerator device.

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

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

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

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

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

AOCL Utilities for Managing an FPGA Board

The following AOCL utility options direct the AOCL to perform tasks such as installing and programming your FPGA board, and running diagnostic tests.

Note: Multiple devices support is a beta feature of the AOCL version 13.1.

program

To configure a new FPGA hardware image onto the accelerator board manually, invoke the `aocl program <your_kernel_filename>.aocx` command.

If you have multiple FPGA devices connected to your system, you must specify the target FPGA device for your kernel. To configure an FPGA hardware image onto a specific FPGA device, invoke the command

`aocl program <device_name> <your_kernel_filename>.aocx`

where <*device_name*> refers to the acl number (e.g. acl0 to acl15) that corresponds to your FPGA device. The <*device_name*> of an FPGA device correlates with the order in which you install your FPGA boards in your system.

flash

If supported, you can initialize your FPGA with a specified startup configuration by invoking the `aocl flash <your_kernel_filename>.aocx` command.

Attention: The AOCL flash utility does not support multiple devices configuration.

install

To install your FPGA accelerator boards and the OS device driver provided in your board package into the current host system, invoke the `aocl install` command.

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

Remember: The board-specific tools are referenced by the *AOCL_BOARD_PACKAGE_ROOT* environment variable.

diagnostic

To run the board vendor's test program for your accelerator board, invoke the `aocl diagnostic` command.

If you want to run diagnostic tests for multiple FPGA devices in your system, you must specify the target FPGA device. To diagnose a specific FPGA device, invoke the command

`aocl diagnostic <device_name>`

where <*device_name*> refers to the acl number (e.g. acl0 to acl15) that corresponds to your FPGA device. The <*device_name*> of an FPGA device correlates with the order in which you install your FPGA boards in your system.

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

Related Information

Altera SDK for OpenCL Getting Started Guide

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

Kernel Programming Considerations

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

Preprocessor Macros on page 1-20

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

__constant Address Space Qualifiers on page 1-22

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

Use Structures Arguments in OpenCL Kernels on page 1-23

Convert each structure parameter to a pointer that points to a structure.

Kernel Pragmas and Attributes on page 1-23

You can increase the data processing efficiency of your OpenCL kernel by specifying kernel pragmas and attributes.

Preprocessor Macros

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

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

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

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

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

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

```
    sum[get_global_id(0)] = accum;
}
```

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

```
aoc -D UNROLL_FACTOR=4 sum.cl
```

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

You can use preprocessor macros to control how the AOC optimizes your kernel without modifying your kernel source code. For example, if you want to compile the same kernel multiple times with required work-group sizes of 64 and 128, you can define a `WORK_GROUP_SIZE` preprocessor macro for the kernel attribute `reqd_work_group_size`, as shown below:

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

Compile the kernel multiple times by typing the following commands:

```
aoc -o myKernel_64.aocx -D WORK_GROUP_SIZE=64 myKernel.cl
aoc -o myKernel_128.aocx -D WORK_GROUP_SIZE=128 myKernel.cl
```

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

Conditional Compilation Based on Preprocessor Macros

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

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

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

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

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

__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 AOC does not support function scope `__constant` variables. Replace function scope `__constant` variables with file scope constant variables. You can also replace function scope `__constant` variables with `__constant` buffers that the host passes to the kernel.

File Scope __constant Variables

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

The AOC supports only scalar file scope constant data. To avoid long compilation times, file scope constant data must not exceed 32 kB in size.

For example, you may set the `__constant` address space qualifier as follows:

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

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

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

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

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

Pointers to __constant Parameters from the Host

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

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

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

Table 1-2: 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 program and your kernel.

Use Structures Arguments in OpenCL Kernels

Convert each structure parameter to a pointer that points to a structure.

The table below describes how you can convert structure parameters:

Table 1-3: Converting Structure Parameters to Pointers that Point to Structures

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

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

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

Kernel Pragmas and Attributes

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

The following pragma is available in the AOCL version 13.1:

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

  When you include the `#pragma unroll <N>` directive, the AOC attempts to unroll the loop at most <*N*> times. Consider the code fragment below. By assigning a value of 2 as an argument to `#pragma unroll`, you direct the AOC to unroll the loop twice.

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

  The AOC might unroll simple loops even if they are not annotated by a pragma. If you do not specify a value for <*N*>, the AOC attempts to unroll the loop fully if it understands the trip count. The AOC issues a warning if it cannot execute the unroll request.

  Attention: Provide an unroll factor whenever possible.

The following kernel attributes are available in the AOCL version 13.1:

For specifying work-group sizes:

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

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

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

- If your kernel contains a barrier, the AOC sets a default maximum work-group size of 256 work-items.
- if your kernel contains a barrier or refers to the local work-item ID, or if you query the work-group size in your host code, the runtime defaults the work-group size to one work-item.
- If your kernel does not contain a barrier or refer to the local work-item ID, or if your host code does not query the work-group size, the runtime defaults the work-group size to the global NDRange size.

For optimizing data processing efficiency:

- `num_compute_units`—specifies the number of compute units the AOC instantiates to process the kernel.

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

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

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

  The increased number of compute units achieves higher throughput at the expense of global memory bandwidth contention among compute units.

- `num_simd_work_items`—specifies the number of work-items within a work-group the AOC executes in a single instruction multiple data (SIMD) manner.

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

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

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

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

For loop unrolling:

- `max_unroll_loops`—specifies the maximum number of times the AOC can unroll all loops in a given kernel automatically.

  You can override this attribute by specifying an unroll factor for an individual loop using the `#pragma unroll` directive.

  For example, if you do not want to unroll a particular loop, include the directive `#pragma unroll 1` for that loop.

  Attention: If you do not specify a `#pragma unroll` value for a loop, the AOC might unroll that loop up to the amount you specify for `max_unroll_loops`.

For resource sharing:

- `max_share_resources`—specifies the maximum number of times the AOC can reuse a shared resource (operator) without reducing computational throughput.
- `num_share_resources`—specifies the number of times the AOC reuses a shared resource (operator).

  The AOC attempts to reuse each resource by at least this amount, which might reduce computational throughput.

For optimizing memory access efficiency:

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

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

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

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

- `task`—identifies the kernel that the AOCL host should execute in a single work-item, as shown in the example below.

  The AOC performs memory dependence analysis on the kernel code to determine the level of pipeline parallelism that the kernel can achieve.

```
__attribute__((task))
__kernel void fft (__global float2 * dataIn, ...)
{
   //statements
   for (i = 0; i < FFT_POINTS; i++)
   {
      //statements
   }
   //statements
}
```

- `buffer_location`—defines the global memory type in which you can allocate a buffer.

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

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

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

Related Information

Altera SDK for OpenCL Optimization Guide

For more information on kernel pragmas and attributes, refer to the *Optimization of Data Processing Efficiency* section of the AOCL Optimization Guide. For more information on heterogeneous memory support, refer to the *Heterogeneous Memory Buffers* section of the AOCL Optimization Guide. For more information on the `task` kernel attribute, refer to the *Single Work-Item Execution* section of the AOCL Optimization Guide.

Host Programming Considerations

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

Host Binary Requirement on page 1-28
When compiling the host application, you must target the x86-64 (64-bit) architecture.

Host Machine Memory Requirements on page 1-28
The machine that runs the host program must have enough host memory to support several components simultaneously.

FPGA Programming on page 1-29
The AOC is an offline compiler that compiles kernels independent of the host application. To load the kernels into the OpenCL runtime, you must use the `clCreateProgramWithBinary` function in your host program.

Multiple Host Threads on page 1-32
The AOCL host library is not thread-safe.

Partitioning Global Memory Accesses on page 1-32
You can invoke the `--sw-dimm-partition` flag of the `aoc` command to partition data into the two memory interfaces of the FPGA board manually.

Partitioning Heterogeneous Global Memory Accesses on page 1-33
For heterogeneous memory systems, you can partition memory buffers manually across banks of the same memory type by including the `--no-interleaving <memory_type>` flag of the `aoc` command.

Out-of-Order Command Queues on page 1-34
The AOC command queues do not support out-of-order command execution.

Modifying Host Program for Structure Parameter Conversion on page 1-34
If you convert any structure parameters to pointers-to-constant structures in your OpenCL kernel, you must modify your host program accordingly.

Host Binary Requirement

When compiling the host application, you must target the x86-64 (64-bit) architecture. The AOCL host runtime does not support x86-32 (32-bit) binaries or any other architecture.

Host Machine Memory Requirements

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

The host machine must support the following components:

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

FPGA Programming

The AOC is an offline compiler that compiles kernels independent of the host application. To load the kernels into the OpenCL runtime, you must use the `clCreateProgramWithBinary` function in your host program.

The following host code demonstrates the usage of the `clCreateProgramWithBinary` function to program an FPGA device:

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

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

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

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

Programming Multiple FPGA Devices

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

Attention: Multiple FPGA devices support is a beta feature of the AOCL version 13.1.

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

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

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

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

Probing the OpenCL FPGA Devices

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

1. To direct the host to identify the number of OpenCL FPGA devices, add the following lines to code to your host application:

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

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

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

Querying Device Information

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

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

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

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

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

where <*N*> is the device number, <*board_name*> is the board designation you use to target your FPGA device when you invoke the `aoc` command, and <*name_of_FPGA_board*> is the advertised name of the FPGA board.

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

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

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

Loading Kernels for Multiple FPGA Devices

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

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

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

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

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

```
    size_t *lengths=(size_t*)malloc(num_devices*sizeof(size_t));
    cl_int err;

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

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

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

// main program

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

Multiple Host Threads

The AOCL host library is not thread-safe.

Partitioning Global Memory Accesses

You can invoke the `--sw-dimm-partition` flag of the `aoc` command to partition data into the two memory interfaces of the FPGA board manually. The data accesses both banks simultaneously, effectively doubling your memory bandwidth.

To partition global memory manually, perform the following tasks:

1. At a command prompt, type the command
`aoc --sw-dimm-partition <your_kernel_filename>.cl` to compile your OpenCL kernels and configure the memory banks as separate address spaces.
2. When you create an OpenCL buffer in your host program, allocate the buffer to one of the two banks with the following `CL_MEM_BANK` flags:
   - Specify `CL_MEM_BANK_1_ALTERA` to allocate the buffer to the lowest available memory region.
   - Specify `CL_MEM_BANK_2_ALTERA` to allocation memory to the second bank (if available).

   For example, the following `clCreateBuffer` call allocates memory into the second bank:

   ```
   clCreateBuffer(context, CL_MEM_BANK_2_ALTERA | CL_MEM_READ_WRITE, size, 0, 0);
   ```

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

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

Related Information

Altera SDK for OpenCL Optimization Guide
For more information on optimizing global memory accesses, refer to the *Optimize Global Memory Accesses* section of the AOCL Optimization Guide.

Partitioning Heterogeneous Global Memory Accesses

For heterogeneous memory systems, you can partition memory buffers manually across banks of the same memory type by including the `--no-interleaving <memory_type>` flag of the `aoc` command.

To partition manually a global memory type in a heterogeneous memory system, perform the following tasks:

1. At a command prompt, type
`aoc --no-interleaving <memory_type> <your_kernel_filename>.cl` to compile your OpenCL kernels and configure the memory bank(s) of the specified memory type as separate address spaces.

   If you want to partition more than one memory type manually, add a `--no-interleaving <memory_type>` flag for each memory type.
2. When you create an OpenCL buffer in your host program, allocate the buffer to one of the banks using the `CL_MEM_HETEROGENEOUS_ALTERA` flag.

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

   ```
   mem = clCreateBuffer(context,
                        flags|CL_MEM_HETEROGENEOUS_ALTERA,
   ```

```
                    memSize,
                    NULL,
                    &errNum);
```

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

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

The `clCreateBuffer` call allocates memory into a certain global memory type based on what you specify in the kernel argument. If a memory (`cl_mem`) object residing in a memory type is set as a kernel argument that corresponds to a different memory technology, the host moves the memory object automatically when it queues the kernel.

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

Related Information

Altera SDK for OpenCL Optimization Guide

For more information on optimizing heterogeneous global memory accesses, refer to the *Heterogeneous Memory Buffers* section of the AOCL Optimization Guide.

Out-of-Order Command Queues

The AOC command queues do not support out-of-order command execution.

Modifying Host Program for Structure Parameter Conversion

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

Perform the following changes to your host program:

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 program no longer needs to call a kernel that uses the structure buffer, release the `cl_mem` buffer.

Support Statuses of OpenCL Features A

2013.12.13

OCL002-13.1.1 Subscribe Send Feedback

The Altera SDK for OpenCL (AOCL) supports the OpenCL Specification version 1.0. The AOCL host runtime conforms with the OpenCL platform layer and application programming interface (API), with clarifications and exceptions.

The following sections outline the support statuses of the OpenCL features described in the *OpenCL Specification version 1.0*.

Related Information

OpenCL Specification version 1.0

OpenCL Programming Language Implementation

OpenCL is based on C99 with some limitations. Section 6 of the *OpenCL Specification version 1.0* describes the OpenCL C programming language. The AOCL conforms with the OpenCL C programming language with clarifications and exceptions. The table below summarizes the support statuses of the features in the OpenCL programming language implementation.

Attention: The support status "●" means that a clarification for the supported feature is available in the Notes column. The support status "○" means that the feature is supported with exceptions identified in the Notes column. A feature that is not supported by the AOCL is identified with an "X". OpenCL programming language implementations that are supported with no additional clarifications are not shown.

Section	Feature	Support Status	Notes
6.1.1	*Built-in Scalar Data Types*		
	double precision float	○	Preliminary support for all double precision float built-in scalar data types. This feature might not conform with the OpenCL Specification version 1.0.
	half	X	
6.1.2	*Built-in Vector Data Types*	○	Preliminary support for vectors with three elements. Three-element vectors might not conform with the OpenCL Specification version 1.0.
6.1.3	*Built-in Data Types*	X	
6.1.4	*Reserved Data Types*	X	

Section	Feature	Support Status	Notes
6.1.5	*Alignment of Types*	●	All scalar and vector types are aligned as required (vectors with three elements are aligned as if they had four elements).
6.2.1	*Implicit Conversions*	●	Refer to Section 6.2.6: *Usual Arithmetic Conversions* in the *OpenCL Specification version 1.2* for an important clarification of implicit conversions between scalar and vector types.
6.2.2	*Explicit Casts*	●	The AOCL allows scalar data casts to a vector with a different element type.
6.5	*Address Space Qualifiers*	○	Function scope `__constant` variables are not supported.
6.6	*Image Access Qualifiers*	X	
6.7	*Function Qualifiers*		
6.7.2	*Optional Attribute Qualifiers*	●	Refer to the *Altera SDK for OpenCL Optimization Guide* for tips on using `reqd_work_group_size` to improve kernel performance. The AOCL parses but ignores the `vec_type_hint` and `work_group_size_hint` attribute qualifiers.
6.9	*Preprocessor Directives and Macros*		
	`#pragma` directive: `#pragma unroll`	●	The AOC supports only `#pragma unroll`. You may assign an integer argument to the unroll directive to control the extent of loop unrolling. For example, `#pragma unroll 4` unrolls four iterations of a loop. By default, an unroll directive with no unroll factor causes the AOC to attempt to unroll the loop fully. Refer to the *Altera SDK for OpenCL Optimization Guide* for tips on using `#pragma unroll` to improve kernel performance.
	`__ENDIAN_LITTLE__` defined to be value `1`	●	The target FPGA is little-endian.
	`__IMAGE_SUPPORT__`	X	`__IMAGE_SUPPORT__` is undefined; the AOCL does not support images.
6.10	*Attribute Qualifiers*—The AOC parses attribute qualifiers as follows:		
6.10.2	*Specifying Attributes of Functions*—Structure-type kernel arguments	X	Convert structure arguments to a pointer to a structure in global memory.
6.10.3	*Specifying Attributes of Variables*—`endian`	X	

Section	Feature	Support Status	Notes
6.10.4	*Specifying Attributes of Blocks and Control-Flow-Statements*	X	
6.10.5	*Extending Attribute Qualifiers*	●	The AOC can parse attributes on various syntactic structures. It reserves some attribute names for its own internal use. Refer to the *Kernel Pragmas and Attributes* section for more information about these attribute names. Refer to the *Altera SDK for OpenCL Optimization Guide* for tips on how to optimize kernel performance using these kernel attributes.
6.11.2	*Math Functions*		
	built-in math functions	○	Preliminary support for built-in math functions for double precision float. These functions might not conform with the OpenCL Specification version 1.0.
	built-in `half_` and `native_` math functions	○	Preliminary support for built-in `half_` and `native_` math functions for double precision float. These functions might not conform with the OpenCL Specification version 1.0.
6.11.5	*Geometric Functions*	○	Preliminary support for built-in geometric functions for double precision float. These functions might not conform with the OpenCL Specification version 1.0. Refer to *Argument Types for Built-in Geometric Functions* for a list of built-in geometric functions supported by the AOCL.
6.11.8	*Image Read and Write Functions*	X	
6.11.9	*Synchronization Functions*—the barrier synchronization function	○	Clarifications and exceptions: If a kernel specifies the `reqd_work_group_size` or `max_work_group_size` attribute, barrier supports the corresponding number of work-items. If neither attribute is specified, a barrier is instantiated with a default limit of 256 work-items. The work-item limit is the maximum supported work-group size for the kernel; this limit is enforced by the runtime.

Section	Feature	Support Status	Notes
6.11.11	*Async Copies from Global to Local Memory, Local to Global Memory, and Prefetch*	○	The implementation is naive: Work-item (0,0,0) performs the copy and the `wait_group_events` is implemented as a barrier. If a kernel specifies the `reqd_work_group_size` or `max_work_group_size` attribute, `wait_group_events` supports the corresponding number of work-items. If neither attribute is specified, `wait_group_events` is instantiated with a default limit of 256 work-items.
Additional built-in vector functions from the *OpenCL Specification version 1.2* Section 6.12.12: *Miscellaneous Vector Functions*:			
	`vec_step`	●	
	`shuffle`	●	
	`shuffle2`	●	
OpenCL Specification version 1.2 Section 6.12.13: *printf*		○	Preliminary support. This feature might not conform with the OpenCL Specification version 1.0. See below for details.

Section	Feature	Support Status	Notes

The `printf` function in OpenCL has syntax and features similar to the `printf` function in C99, with a few exceptions. For details, refer to the *OpenCL Specification version 1.2*.

To use a `printf` function, there are no requirements for special compilation steps, buffers, or flags. You can compile kernels that include `printf` instructions with the usual `aoc` command.

During kernel execution, `printf` data is stored in a global `printf` buffer that the AOC allocates automatically. The size of this buffer is 64 kB; the total size of data arguments to a `printf` call should not exceed this size. When kernel execution completes, the contents of the `printf` buffer are printed to standard output.

Buffer overflows are handled seamlessly; `printf` instructions can be executed an unlimited number of times. However, if the `printf` buffer overflows, kernel pipeline execution stalls until the host reads the buffer and prints the buffer contents.

Because `printf` functions store their data into a global memory buffer, the performance of your kernel will drop if it includes such functions.

There are no usage limitations on `printf` functions. You can use `printf` instructions inside `if`-then-`else` statements, loops, etc. A kernel can contain multiple `printf` instructions executed by multiple work-items.

Format string arguments and literal string arguments of `printf` calls are transferred to the host system from the FPGA using a special memory region. This memory region can overflow if the total size of the `printf` string arguments is large (3000 characters or less is usually safe in a typical OpenCL application). If there is an overflow, the error message `cannot parse auto-discovery string at byte offset 4096` is printed during host program execution.

Output from `printf` is never intermixed, even though work-items may execute `printf` functions concurrently. However, the order of concurrent `printf` execution is not guaranteed. In other words, `printf` outputs might not appear in program order if the `printf` instructions are in concurrent datapaths.

Related Information

- **Kernel Pragmas and Attributes** on page 1-23
- **Altera SDK for OpenCL Optimization Guide**
- **OpenCL Specification version 1.2**

OpenCL Programming Language Restrictions

The AOCL conforms with the OpenCL Specification restrictions on specific programming language features, as described in section 6.8 of the *OpenCL Specification version 1.0*.

Warning: The AOC does not enforce restriction on certain disallowed programming language features. You must ensure that your kernel code does not contain features that are not supported by the OpenCL Specification version 1.0.

Feature	Support Status	Notes
pointer assignments between address spaces	●	Arguments to `__kernel` functions declared in a program that are pointers must be declared with the `__global`, `__constant`, or `__local` qualifier. The AOC enforces the OpenCL restriction against pointer assignments between address spaces.
pointers to functions	X	The AOC does not enforce this restriction.
structure-type kernel arguments	X	Convert structure arguments to a pointer to a structure in global memory.
images	X	The AOCL does not support images.
bit fields	X	The AOC does not enforce this restriction.
variable length arrays and structures	X	
variable macros and functions	X	
C99 headers	X	
`extern`, `static`, `auto`, and `register` storage-class specifiers	X	The AOC does not enforce this restriction.
predefined identifiers	●	Use the `-D` option of the `aoc` command to provide preprocessor symbol definitions in your kernel code.
recursion	X	The AOC does not enforce this restriction.
irreducible control flow	X	The AOC does not enforce this restriction.
writes to memory of built-in types less than 32 bits in size	○	Store operations less than 32 bits in size might result in lower memory performance.
declaration of arguments to `__kernel` functions of type `event_t`	X	The AOC does not enforce this restriction.
elements of a `struct` or a `union` belonging to different address spaces	X	The AOC does not enforce this restriction. **Warning:** Assigning elements of a `struct` or a `union` to different address spaces might cause a fatal error.

Argument Types for Built-in Geometric Functions

The AOCL supports scalar and vector argument built-in geometric functions with certain limitations.

Function	Argument Type	
	float	double
`cross`	•	•
`dot`		•
`distance`		•
`length`		•
`normalize`		•
`fast_distance`		—
`fast_length`		—
`fast_normalize`		—

Numerical Compliance Implementation

Section 7 of the *OpenCL Specification version 1.0* describes features of the C99 and IEEE 754 standards that the OpenCL compliant devices must support. The AOCL operates on 32-bit and 64-bit floating-point values in IEEE Standard 754-2008 format, but not all floating-point operators have been implemented.

The table below summarizes the implementation statuses of the floating-point operators:

Section	Feature	Support Status	Notes
7.1	*Rounding Modes*	○	Conversion between integer and single and half precision floating-point types support all rounding modes. Conversions between integer and double precision floating-point types support all rounding modes on a preliminary basis. This feature might not conform with the OpenCL Specification version 1.0.
7.2	*INF, NaN and Denormalized Numbers*	○	Infinity (INF) and Not a Number (NaN) results for single precision operations are generated in a manner that conforms with the OpenCL Specification version 1.0. Most operations that handle denormalized numbers are flushed prior to and after a floating-point operation. Preliminary support for double precision floating-point operation. This feature might not conform with the OpenCL Specification version 1.0.
7.3	*Floating-Point Exceptions*	X	

Section	Feature	Support Status	Notes
7.4	*Relative Error as ULPs*	○	Single precision floating-point operations conform with the numerical accuracy requirements for an embedded profile of the OpenCL Specification version 1.0. Preliminary support for double precision floating-point operation. This feature might not conform with the OpenCL Specification version 1.0.
7.5	*Edge Case Behavior*	●	

Image Addressing and Filtering Implementation

The AOCL does not support image addressing and filtering. The AOCL does not support images.

Atomic Functions

Section 9 of the *OpenCL Specification version 1.0* describes a list of optional features that some OpenCL implementations might support. The AOCL supports atomic functions conditionally.

- Section 9.5: *Atomic Functions for 32-bit Integers*—The AOCL supports all 32-bit global and local memory atomic functions. The AOCL also supports 32-bit atomic functions described in Section 6.11.11 of the *OpenCL Specification version 1.1* and Section 6.12.11 of the *OpenCL Specification version 1.2*.
    - The AOCL does not support 64-bit atomic functions described in Section 9.7 of the *OpenCL Specification version 1.0*.

Attention: The use of atomic functions might lower the performance of your design. The operating frequency of the hardware might decrease further if you implement more than one type of atomic functions in the kernel.

Embedded Profile Implementation

Section 10 of the *OpenCL Specification version 1.0* describes the OpenCL embedded profile. The AOCL conforms with the OpenCL embedded profile with clarifications and exceptions.

The table below summarizes the clarifications and exceptions to the OpenCL embedded profile:

Clause	Feature	Support Status	Notes
1	64-bit integers	●	
2	3D images	X	The AOCL does not support images.
3	Create 2D and 3D images with `image_channel_data_type` values	X	The AOCL does not support images.

Clause	Feature	Support Status	Notes
4	Samplers	X	
5	Rounding modes	●	The default rounding mode for `CL_DEVICE_SINGLE_FP_CONFIG` is `CL_FP_ROUND_TO_NEAREST`.
6	Restrictions listed for single precision basic floating-point operations	X	
7	half type	X	This clause of the OpenCL Specification version 1.0 does not apply to the AOCL.
8	Error bounds listed for conversions from `CL_UNORM_INT8`, `CL_SNORM_INT8`, `CL_UNORM_INT16` and `CL_SNORM_INT16` to float	●	Refer to the table below for a list of allocation limits.

AOCL Allocation Limits

Item	Limit
Maximum number of contexts	4
Maximum number of queues per context	28
Maximum number of program objects per context	20
Maximum number of event objects per context	16000
Maximum number of dependencies between events within a context	1000
Maximum number of event dependencies per command	20
Maximum number of concurrently running kernels	The total number of queues
Maximum number of enqueued kernels	1000
Maximum number of kernels per FPGA device	32
Maximum number of arguments per kernel	128
Maximum total size of kernel arguments	256 bytes per kernel

Document Revision History B

Date	Version	Changes

December 2013	13.1.1	• Removed the section *-W and -Werror*, and replaced it with two sections: *-W* and *-Werror*.
		• Updated the following contents to reflect multiple devices support:
		  • The figure *The AOCL FPGA Programming Flow*.
		  • *--list-boards* section.
		  • *-board <board_name>* section.
		  • *AOCL Utilities for Managing an FPGA Board* section.
		  • Added the subsection *Programming Multiple FPGA Devices* under *FPGA Programming*.
		• The following contents were added to reflect heterogeneous global memory support:
		  • *--no-interleaving* section.
		  • `buffer_location` kernel attribute under *Kernel Pragmas and Attributes*.
		  • *Partitioning Heterogeneous Global Memory Accesses* section.
		• Modified support status designations in *Appendix: Support Statuses of OpenCL Features*.
		• Removed information on OpenCL programming language restrictions from the section *OpenCL Programming Language Implementation*, and presented the information in a new section titled *OpenCL Programming Language Restrictions*.

Date	Version	Changes

November 2013	13.1.0	• Reorganized information flow.
		• Updated and renamed *Altera SDK for OpenCL Compilation Flow* to *AOCL FPGA Programming Flow*.
		• Added figures *One-Step AOC Compilation Flow* and *Two-Step AOC Compilation Flow*.
		• Updated the section *Contents of the AOCL Version 13.1*.
		• Removed the following sections:
		  • *OpenCL Kernel Source File Compilation*.
		  • *Using the Altera Offline Kernel Compiler*.
		  • *Setting Up Your FPGA Board*.
		  • *Targeting a Specific FPGA Board*.
		  • *Running Your OpenCL Application*.
		  • *Consolidating Your Kernel Source Files*.
		  • *Aligned Memory Allocation*.
		  • *Programming the FPGA Hardware*.
		  • *Programming the Flash Memory of an FPGA*.
		• Updated and renamed *Compiling the OpenCL Kernel Source FIle* to *AOC Compilation Flows*.
		• Renamed *Passing File Scope Structures to OpenCL Kernels* to *Use Structure Arguments in OpenCL Kernels*.
		• Updated and renamed *Augmenting Your OpenCL Kernel by Specifying Kernel Attributes and Pragmas* to *Kernel Pragmas and Attributes*.
		• Renamed *Loading Kernels onto an FPGA* to *FPGA Programming*.
		• Consolidated *Compiling and Linking Your Host Program*, *Host Program Compilation Settings*, and *Library Paths and Links* into a single section.
		• Inserted the section *Preprocessor Macros*.
		• Renamed *Optimizing Global Memory Accesses* to *Partitioning Global Memory Accesses*.

Date	Version	Changes

June 2013	13.0 SP1.0	• Added the section *Setting Up Your FPGA Board*.
		• Removed the subsection *Specifying a Target FPGA Board* under *Kernel Programming Considerations*.
		• Inserted the subsections *Targeting a Specific FPGA Board* and *Generating Compilation Reports* under *Compiling the OpenCL Kernel Source File*.
		• Renamed *File Scope __constant Address Space Qualifier* to *__constant Address Space Qualifiers*, and inserted the following subsections:
		  • *Function Scope __constant Variables*.
		  • *File Scope __constant Variables*.
		  • *Points to __constant Parameters from the Host*.
		• Inserted the subsection *Passing File Scope Structures to OpenCL Kernels* under *Kernel Programming Considerations*.
		• Renamed *Modifying Your OpenCL Kernel by Specifying Kernel Attributes and Pragmas* to *Augmenting Your OpenCL Kernel by Specifying Kernel Attributes and Pragmas*.
		• Updated content for the `unroll` pragma directive in the section *Augmenting Your OpenCL Kernel by Specifying Kernel Attributes and Pragmas*.
		• Inserted the subsections *Out-of-Order Command Queues* and *Modifying Host Program for Structure Parameter Conversion* under *Host Programming Considerations*.
		• Updated the sections *Loading Kernels onto an FPGA Using clClreateProgramWithBinary* and *Aligned Memory Allocation*.
		• Updated flash programming instructions.
		• Renamed *Optional Extensions* in *Appendix B* to *Atomic Functions*, and updated its content.
		• Removed *Platform Layer and Runtime Implementation* from *Appendix B*.

May 2013	13.0.1	• Explicit memory fence functions are now supported; the entry is removed from the table OpenCL Programming Language Implementation. • Updated the section Programming the Flash Memory of an FPGA. • Added the section *Modifying Your OpenCL Kernel by Specifying Kernel Attributes and Pragmas* to introduce kernel attributes and pragmas that can be implemented to optimize kernel performance. • Added the section Optimizing Global memory Accesses to discuss data partitioning. • Removed the section *Programming the FPGA with the aocl program Command* from Appendix A.

Date	Version	Changes
May 2013	13.0.0	• Updated compilation flow. • Updated kernel compiler commands. • Included Altera SDK for OpenCL Utility commands. • Added the section *OpenCL Programming Considerations*. • Updated flash programming procedure and moved it to *Appendix A*. • Included a new `clCreateProgramWithBinary` FPGA hardware programming flow. • Moved the hostless `clCreateProgramWithBinary` hardware programming flow to *Appendix A* under the title *Programming the FPGA with the aocl program Command*. • Moved updated information on allocation limits and OpenCL language support to *Appendix B*.
November 2012	12.1.0	Initial release.