AOCC User Guide

Publication #	**57222** Revision: **3.1**
Issue Date:	**July 2021**

Contents

List of Figures

List of Tables

Revision History

Date	Revision	Description
July 2021	3.1	• Updated Chapter 4. • Performed some general edits.
March 2021	3.0	Initial version.

Chapter 1 Introduction

AMD Optimizing C/C++ Compiler (AOCC) is a highly optimized C, C++, and Fortran compiler for x86 targets, especially for Zen based AMD processors. It supports Flang as the default Fortran front-end compiler. This document does not cover the internals of Low-Level Virtual Machine (LLVM) or AOCC.

AOCC 3.1 release is based on LLVM 12 release (llvm.org, 14th April 2021) with Flang as a Fortran front-end added with F2008 and Real 128 features. It is an incremental version of AOCC 3.0 that includes bug fixes and a support for compiler directives in Flang. For more information, refer *User Guide* and *Flang Guide*.

Chapter 2 Programming Language Support

LLVM is a compiler infrastructure covering many programming languages and target processors. AOCC focuses on the following:

- C, C++, and Fortran programming languages
- Target dependent (x86 targets especially AMD processors) and target independent optimizations

AOCC leverages LLVM Clang, which is the compiler and driver for C and C++ programs. Flang is the compiler and driver for the Fortran programs.

2.1 C and C++ Programs

Clang is not just a C and C++ frontend, which compiles the program to LLVM intermediate representation (IR). It is also the driver, which ensures that it invokes all the required LLVM optimization passes and targets code generation to generating the binaries. You can use Clang as an end-to-end driver or if you are an advanced user, you can do a multi-pass compilation calling for each of the compilation phases manually. The AOCC package comes with all the tools and libraries essential for using clang as an end-to-end driver or to perform a manual multi-pass compilation.

2.2 Fortran Programs

Flang is the Fortran frontend designed for an integration with LLVM and is suitable for an interoperability with Clang/LLVM. Flang consists of the following two components:

- flang1: It is invoked by the front-end driver, which is responsible for transforming the Fortran programs into tokens. Then, the parser transforms these tokens into Abstract Syntax Tree (AST). This AST is then transformed into canonical form, which is used to generate the ILM code.
- flang2: It picks up the ILM code from flang1 and transforms it into ILI, which is then optimized by the internal optimizer. The optimized ILI is then transformed into LLVM IR. The frontend driver transfers this LLVM IR to the LLVM optimizer for optimization and target code generation.

Note: *AOCC Flang extends the GitHub version with enhancements and stability.*

Chapter 3 Support

If you encounter any issues with this release or need assistance using the compiler, you must log a ticket in the *AMD support portal* using your AMD SSO login.

Note: *If required, the project name is CPUPerfCompiler.*

This support channel between AMD and you is 1:1. So, you can feel free to share programs or code snippets to help us assist you better. AMD assures you that these discussions and code snippets will strictly remain confidential. They will not be shared with anyone else under any circumstances.

3.1 Logging a Ticket

Complete the following steps to log a ticket in the AMD support portal:

1. Click the **Create** button on the top of the Web page.
2. Please select the following options from the respective drop-down lists:
   - **Project:** CPUPerfCompiler (CPUP)
   - **Issue Type:** Bug/New Feature (for feature request)/Task (for Support)
3. Provide clear reproducible steps and a test case.

   Note: *You can provide us the code to help us serve you better.*
4. Mention the output of command `clang -v` (provide the AOCC version you are using).
5. Select the severity based on the impact.

3.2 Creating an AMD SSO Login

Complete the following steps to create an AMD SSO login:

1. Access the URL *http://ontrack.amd.com/projects/CPUP*.

   The login page is displayed as follows:

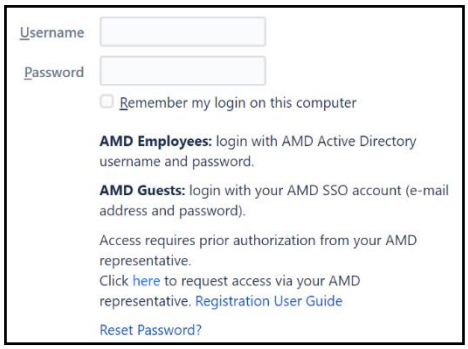

   Figure 1. Login Page

2. Click **here**.

   The SSO registration page is displayed.

3. Fill in all the mandatory details and click **Register**.

   Note: *Provide the AMD contact person's name and email ID accurately. He/she will initiate the process in AMD based on your NDA agreement.*

Chapter 4 Using AOCC

4.1 Prerequisite

The AOCC binaries can run optimally only on the Linux systems using *glibc* version 2.17 and later.

4.2 AOCC Optimizer

AOCC includes many optimizations independent and dependent targets. A few of these are made default when you use an optimization level 03 and above. You can read more about these in the command line option section. A few other optimizations need a whole program analysis. Hence, they are enabled under Link Time Optimization (LTO) using -flto. The AOCC preferred linker is LLD. Refer the section *4.3.3* for using LLD in the complier driver.

4.3 Using the Compiler

4.3.1 Clang and Clang++

To build and run a C or C++ program, execute the following commands:

```
$ clang [command line flags] xyz.c -o xyz.out
$ ./xyz.out

$ clang++ [command line flags] xyz.cpp -o xyz.out
$ ./xyz.out
```

4.3.2 Flang

To build and run Fortran programs, execute the following commands:

```
$ flang [command line flags] xyz.f90 -o xyz.out
$ ./xyz.out
```

4.3.3 LLD Linker

To use an LLD linker, execute the following commands:

```
$ clang [command line flags] -fuse-ld=lld xyz.c abc.c -o xyz.out [here -fuse-ld=lld is
optional as this option is default]
$ ./xyz.out
```

4.4 Libraries

Some applications will benefit from the optimized libraries. AOCC will work seamlessly with these libraries. It is recommended that you evaluate these libraries while building your application with AOCC. They will boost the performance of your application over the compiler optimizations that come with AOCC. For more information on AMD Optimizing CPU Libraries (AOCL), refer *https://developer.amd.com/amd-aocl/*.

4.4.1 Configuring Library Path

Execute the following command to configure the library path:

- For 64-bit Library
  `export LD_LIBRARY_PATH=<compdir>/aocc-compiler-<ver>/lib:$LD_LIBRARY_PATH`
- For 32-bit Library
  `export LD_LIBRARY_PATH=<compdir>/aocc-compiler-<ver>/lib32:$LD_LIBRARY_PATH`
- For other AMD optimizing CPU libraries
  `export LD_LIBRARY_PATH=<Path to AMD optimizing CPU Libraries>:$LD_LIBRARY_PATH`

4.4.2 Generate Vector Library Calls

Execute one of the following commands to generate the vector library calls from AOCC:

- `$ clang [command line flags] xyz.c -fveclib=AMDLIBM -o xyz.out`
- `$ clang [command line flags] xyz.c -mllvm -vector-library=AMDLIBM -o xyz.out`

4.4.3 Linker Option

Execute the following command to link AMDLIBM with the linker:

```
$ clang [command line flags] xyz.c -L<compdir>/aocc-compiler-<ver>/lib -lalm -o xyz.out
```

Execute the following command to link other AMD optimizing CPU libraries with linker:

```
$ clang [command line flags] xyz.c -L<Path to AMD optimizing CPU Libraries> -l<library name>
-o xyz.out
```

4.5 OpenMP Debugging Support (OMPD)

The AOCC installation includes OMPD for debugging C/C++ OpenMP programs through a gdb plugin with limited functionality.

Note: *Debugging the code that runs on an offloading device is not supported.*

Complete the following steps to use OMPD for debugging C/C++ OpenMP programs through a *gdb* plugin:

Note: *For using the OMPD plugin, Python 3.5 or later is required.*

1. Add folders **ompd** and **lib** to your **LD_LIBRARY_PATH** using this command:

```
$ export LD_LIBRARY_PATH=<compdir>/aocc-compiler-<ver>/ompd:<compdir>/aocc-compiler-
<ver>/lib:$LD_LIBRARY_PATH
```

2. Set OMP_DEBUG to enabled.

```
$ export OMP_DEBUG=enabled
```

3. Compile the program to be debugged with **-g** and **-fopenmp** options as follows for a sample C source file *xyz.c*:

```
$ <compdir>/aocc-compiler-<ver>/bin/clang -g -fopenmp xyz.c -o xyz.out
```

   Note: *The program to be debugged needs to have a dynamic link dependency on 'libomp.so' under <compdir>/aocc-compiler-<ver>/lib for OpenMP-specific debugging to work correctly. The user can check this using ldd on the generated binary, that is xyz.out.*

4. Debug the binary *xyz.out* by invoking *gdb* with the plugin as follows:

```
$ gdb -x <compdir>/aocc-compiler-<ver>/ompd/__init__.py ./xyz.out
```

   Note: *The plugin <compdir>/aocc-compiler-<ver>/ompd/__init__.py must be used.*

4.5.1 OMPD Commands

The following table describes the OMPD commands:

Table 1. OMPD Commands

Command	Description
`help ompd`	It lists the subcommands available for OpenMP specific debugging.
`ompd init`	• It must be run first to load the libompd.so available in the $LD_LIBRARY_PATH environment variable and to initialize the OMPD library. • It starts the program run and the program stops at a temporary breakpoint at the OpenMP internal location ompd_dll_locations_valid(). • You can continue from the temporary breakpoint for debugging. • You can place breakpoints at the OpenMP internal locations ompd_bp_thread_begin and ompd_bp_thread_end to catch the begin and end events • ompd_bp_task_begin and ompd_bp_task_end breakpoints can be used to catch the beginning and ending of the events • ompd_bp_parallel_begin and ompd_bp_parallel_end can be used to catch the beginning and ending of the parallel events.

4.5.2 OMPD Subcommands

The following table lists the OMPD subcommands that can used inside gdb:

Table 2. OMPD Subcommands

Subcommand	Description
`ompd init`	Finds and initializes the OMPD library.
`ompd bt`	Used to turn the filter *on* or *off* for the bt output on or off. You must specify the *on continued* option to trace the worker threads back to the master threads.
`ompd icvs`	Displays the values of the Internal Control Variables.
`ompd parallel`	Displays the details of the current and enclosing parallel regions.
`ompd step`	Executes step and skip runtime frames as much as possible.
`ompd threads`	Provides the details of the current threads.

4.6 Command Line Options

- Clang - the C, C++ Compiler - Refer Options section in *Clang – the C, C++ Compiler* document
- Flang - the Fortran Compiler - Refer Options section in *Flang – the Fortran Compiler* document