AOCC User Guide

Publication # **57222**	Revision # **3.2**
Issue Date **December 2021**	

Trademarks

AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc.

Dolby is a trademark of Dolby Laboratories.

ENERGY STAR is a registered trademark of the U.S. Environmental Protection Agency.

HDMI is a trademark of HDMI Licensing, LLC.

HyperTransport is a licensed trademark of the HyperTransport Technology Consortium.

Linux® is the registered trademark of Linus Torvalds in the U.S. and other countries.

LLVM™ is a trademark of LLVM Foundation.

Microsoft, Windows, Windows Vista, Windows Server, Visual Studio, and DirectX are registered trademarks of Microsoft Corporation.

MMX is a trademark of Intel Corporation.

OpenCL is a trademark of Apple Inc. used by permission by Khronos.

PCIe is a registered trademark of PCI-Special Interest Group (PCI-SIG).

Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

Dolby Laboratories, Inc.

Manufactured under license from Dolby Laboratories.

Rovi Corporation

This device is protected by U.S. patents and other intellectual property rights. The use of Rovi Corporation's copy protection technology in the device must be authorized by Rovi Corporation and is intended for home and other limited pay-per-view uses only, unless otherwise authorized in writing by Rovi Corporation.

Reverse engineering or disassembly is prohibited.

USE OF THIS PRODUCT IN ANY MANNER THAT COMPLIES WITH THE MPEG-2 STANDARD IS EXPRESSLY PROHIBITED WITHOUT A LICENSE UNDER APPLICABLE PATENTS IN THE MPEG-2 PATENT PORTFOLIO, WHICH LICENSE IS AVAILABLE FROM MPEG LA, L.L.C., 6312 S. FIDDLERS GREEN CIRCLE, SUITE 400E, GREENWOOD VILLAGE, COLORADO 80111.

Contents

List of Tables

Revision History

Date	Revision	Description
December 2021	3.2	• Updated Chapter 5 with Flang sanitization command. • Updated the Chapter 4 with (NO)FREEFORM pragma related information.
July 2021	3.1	Incremental updates and some major/general edits.
March 2021	3.0	Initial version.

Chapter 1 Introduction

The AMD Optimizing C/C++ and Fortran Compiler (AOCC) is highly optimized for x86 targets, especially for AMD “Zen”-based processors. This guide describes how to use AOCC.

AOCC 3.2 is based on the LLVM™ 13 compiler infrastructure (llvm.org, 4 October 2021) and includes bug fixes and support for other new features. For more information, refer the AOCC 3.2 release notes.

Chapter 2 Programming Language Support

AOCC is a high-performance x86 CPU compiler for C, C++, and Fortran programming languages. It supports target-dependent (x86 targets especially AMD processors) and target-independent optimizations.

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

2.1 C, C++, AND FORTRAN Programming Languages

AOCC Clang and AOCC Flang support the preprocessing, parsing, optimization, code generation, assembly, and linking. Using these drivers, overall execution of other tools, such as the compiler, assembler, and linker can be controlled depending on which high-level mode setting is passed. While Clang and Flang are highly integrated, it is important to understand the stages of compilation. These stages are executed in the following sequence:

1. **Driver**

   Clang is not just a C and C++ front-end that compiles the program to LLVM intermediate representation (IR). Clang is also the driver that ensures use of the required LLVM optimization passes and targets code generation to generating the binaries.

   Similar to Clang, Flang is the Fortran front-end compiler and consists of the following two components:

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

   For simplicity, you can use these Clang and Flang as an end-to-end driver. However, for advanced compilation, you can manually execute each compilation phase.

2. **Preprocessing**

   This stage handles the tokenization of the input source file, macro expansion, #include expansion, and handling of the other preprocessor directives. The output of this stage is typically called a *.i* (for C), *.ii* (for C++), or *.i* (for Fortran) file.

3. **Parsing and Semantic Analysis**

   This stage parses the input file, translating the preprocessor tokens into a parse tree. When in the form of a parse tree, it applies semantic analysis to the compute types for expressions and to determine whether the code is well-formed. This stage is responsible for generating most of the compiler warnings and parse errors. The output of this stage from Clang is an Abstract Syntax

Tree (AST). Using Flang, Flang1 will be invoked to transform the program tokens into AST, then into canonical form- that is used to generate ILM code.

4. **LLVM IR Code Generation**

   In Clang, this stage translates an AST into a Low-level Intermediate Code (LLVM IR)

   Using Flang, Flang2 takes up the ILM code generated by Flang1 and transforms it into ILI, which is then optimized by the internal optimizer and then transformed into LLVM IR.

5. **AOCC Optimizer**

   This phase is responsible for optimizing the generated LLVM IR and handling the target-specific code generation. The output of this stage is typically called a *.s* or an assembly file.

6. **Machine Code Generation**

   This phase performs the target specific code generation from the optimized LLVM IR. The output of this stage is typically called a *.s* or assembly file. Clang and Flang also support the use of an integrated assembler from which the code generator produces object files directly. This avoids the overhead of generating the *.s* file and then calling the target assembler.

7. **Assembler**

   This stage runs the target assembler to translate the output of the compiler into a target object file. The output of this stage is typically called a *.o* or object file.

8. **Linker**

   This stage runs the target linker to merge multiple object files into an executable or dynamic library. The output of this stage is typically called an *a.out*, *.dylib*, or *.so* file.

2.2 Compatibility/Conformance to Standards

AOCC supports the following language and debugging standards:

- C: C99, C11, and C17 standards
- C++: C++ 98, C++ 03, C++ 11, C++ 17, and C++ 20 standard[1]
- Fortran: F77, F90, F95, F2003, and F2008 standards[2]
- OMP 4.5 and OMP 5.0 standards for C/C++ programming[3]
- OMP 4.5 standards for Fortran programming
- DWARF 5 standards for C, C++, and Fortran debuggability

1. Refer *https://clang.llvm.org/cxx_status.html#cxx20 for C++ 20* conformance.
2. AOCC does not support F2008 coarrays.
3. Refer *https://releases.llvm.org/13.0.0/tools/clang/docs/OpenMPSupport.html#openmp-implementation-details* for OpenMP 5.0 conformance.

2.2.1 Support for Annex F (IEEE-754/IEC 559) of C99/C11

Clang compiler does not support IEC 559 math functionality. Clang does not control and honor the definition of __STDC_IEC_559__ macro. Under specific options, such as -Ofast and -ffast-math, Clang will enable a range of optimizations that provide faster mathematical operations that may not conform to the IEEE-754 specifications. The macro __STDC_IEC_559__ value may be defined but ignored when these faster optimizations are enabled.

2.2.2 IEEE-754 Support

The Flang compiler does not conform to IEEE-754 specifications when `-Ofast` or `-ffast-math` options are specified. The compiler will enable a range of optimizations that provide faster mathematical operations under `-Ofast` and `-ffast-math` mode of compilation.

Note: *AOCC Flang extends the GitHub version (https://github.com/flang-compiler/flang.git) with enhancements and stability.*

Chapter 3 Working with AOCC

3.1 Installing On Linux

3.1.1 Prerequisites

The following software packages must be installed prior to the AOCC installation:

Table 1. Prerequisites

Package Name	Version(s)	Notes
GCC	5.1.0 or later	C/C++ compiler
libstdc++	6 or later	GNU Standard C++ Library V3
libncurses-dev	5.9 or later	Provides libtinfo, which is a low level terminfo library
zlib	1.2.7 or later	Compression library
Libxml2	2 or later	Parses the XML documents
libquadmath	4.8 or later	GCC Quad-Precision Math Library
python	3.x	Python library

Notes:

1. *For a better performance, it is recommended to use the latest versions of Glibc and Binutils.*
2. *AOCC compiler binaries are suitable only for the Linux® systems having Glibc version 2.17 or later.*

3.1.2 Installation

Note: *This installation does not require root or sudo permission.*

To install *aocc-compiler-<ver>.tar*, execute the following commands:

1. `cd <compdir>`
2. `tar -xvf aocc-compiler-<ver>.tar`
3. `cd aocc-compiler-<ver>`
4. `bash install.sh`

   It will install the compiler and display the AOCC setup instructions.
5. `source <compdir>/setenv_AOCC.sh`

   This will setup the shell environment for using AOCC C, C++, and Fortran compiler where the command is executed.

You must ensure the following:

- Run the bash command `<compdir>/aocc-compiler-<ver>/AOCC-prerequisites-check.sh` to check if you have all the prerequisites and your shell environment is configured correctly.
  - If there are failing checks, correct them (repeat any of the above steps that you may have missed) and `run prerequisites_check.sh` again.
  - Repeat until `AOCC-prerequisites-check.sh` displays **Check:PASSED**.

  Note: *You could proceed if the packages mentioned in the warnings during the failing checks are not required for your run.*
- The compiler is installed and your environment is configured to the current release of AOCC. At any point, you can execute the command source `<compdir>/setenv_AOCC.sh` to set the environment variables for the installed compiler.

3.1.3 SPACK Support

Notes:

1. *SPACK support is available starting from AOCC 2.2.*
2. *In the following steps, <Version Number> in aocc@<Version Number> implies the AOCC version. For example, if you are installing AOCC 3.2.0, you must use aocc@3.2.0.*

Installing AOCC in SPACK

Complete the following steps to installing AOCC compiler in SPACK:

1. Install AOCC:

```
$ spack install -v aocc@<Version Number> +license-agreed
```

2. Add AOCC to the SPACK compiler list:

```
$ spack cd -i aocc@<Version Number>
$ spack compiler add $PWD
$ spack cd -i aocc@<Version Number>
$ spack compiler add $PWD
```

3. List all the available compilers:

```
$ spack compilers
```

Uninstalling AOCC Compiler from SPACK

1. Uninstall AOCC:

```
$ spack uninstall aocc@<Version Number>
```

2. Remove the compiler from the *compiler.yaml* file:

```
$ spack compiler remove aocc@<Version Number>
```

For more details on AOCC in SPACK, refer AMD Developer Central (*https://developer.amd.com/spack/amd-optimized-c-cpp-compiler/*).

3.1.4 Upgrading AMD LibM (ALM)

This is required only when you are upgrading AMD LibM from the AMD portal (*https://developer.amd.com/amd-aocl/*).

Complete the following steps to perform an upgrade:

1. Extract the latest AMD LibM package.
2. Overwrite *aocc-compiler-<ver>/lib/libalm.so* and *aocc-compiler-<ver>/lib/libalm.a* with the latest versions of *libalm.so* and *libalm.a* respectively.
3. Similarly, overwrite *aocc-compiler-<ver>/include/amdlibm.h* and *amdlibm_vec.h* with the latest versions of *amdlibm.h* and *amdlibm_vec.h* respectively.

3.1.5 Supported Operating Systems (OS)

The following OS are supported in this release:

- RHEL 8.x
- SLES 15
- Ubuntu 20.04 LTS
- CentOS 8.x
- Other Linux flavors/versions with glibc 2.17 or higher

3.1.6 Known Issues and Limitations

This release has the following known issues and limitations:

- AOCC binaries can run optimally only on Linux systems having glibc version 2.17 or later.
- Currently, Flang supports only 64-bit targets.

3.2 Invoking AOCC

To set the required environment before invoking compiler driver:

```
$ source <compdir>/setenv_AOCC.sh
```

3.2.1 AOCC Optimizer

AOCC includes many optimizations for independent and dependent targets. Specific optimizations are made default when you use an optimization level *O3* and above. You can read more about these in the command line option section. Some optimizations need a whole program analysis and are enabled under Link Time Optimization (LTO) using -flto. The AOCC preferred linker is LLD. Refer the section LLD Linker for using LLD in the complier driver.

3.2.2 Using the Compiler

3.2.2.1 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
```

3.2.2.2 Flang

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

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

3.2.2.3 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
```

3.2.3 Libraries

Some applications may perform better using the AMD Optimizing CPU Libraries (AOCL). AOCC will work seamlessly with these libraries. It is recommended that you evaluate these libraries while building your application with AOCC. For more information on AMD Optimizing CPU Libraries (AOCL), refer AMD Developer Central (*https://developer.amd.com/amd-aocl/*).

3.2.3.1 Configuring Library Path

Execute the following commands 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
  ```

3.2.3.2 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
```

3.2.3.3 Linking AMD Library

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
```

Chapter 4 Using Pragma Directives

4.1 Flang

Following are the pragma directives specific only to Flang:

4.1.1 NOINLINE

This directive instructs the compiler not to inline the specified routine.

!DIR$ NOINLINE

To use this directive, compiler optimization level should be in -O0 to -O3. The NOINLINE directive overrides the compiler options `-finline-functions` and `-fno-inline-functions`.

Example:

```
!DIR$ NOINLINE
SUBROUTINE func_noinline
    INTEGER :: i
    do i = 0, 5
            WRITE(*, *) "Hello World"
    end do
END SUBROUTINE func_noinline

PROGRAM test_inline
    IMPLICIT NONE
    call func_noinline
END PROGRAM test_inline
```

4.1.2 FORCEINLINE

This directive instructs compiler to always inline the specified routine

!DIR$ FORCEINLINE

To use this directive, compiler optimization level should be in -O0 to -O3. The FORCEINLINE directive overrides the compiler options `-finline-functions` and `-fno-inline-functions.`

Example:

```
!DIR$ FORCEINLINE
SUBROUTINE func_forceinline
    INTEGER :: i
    do i = 0, 5
            WRITE(*, *) "Hello World"
    end do
END SUBROUTINE func_forceinline

PROGRAM test_inline
    IMPLICIT NONE
    call func_forceinline
END PROGRAM test_inline
```

4.1.3 UNROLL

This directive instructs the compiler about the number of times the loop should be unrolled.

!DIR$ UNROLL [(n)]

- **n** – optional parameter, integer constant ranges from 1 - 512
- When **n** equals 0, compiler will decide if unrolling should happen or not

To use this directive, compiler optimization level should be -O1 or above.

If **n** is specified, the optimizer unrolls the loop by n times.

If **n** is not specified or out of range, the optimizer unrolls the loop based on profitability.

Example:

```
Example 1:
subroutine func1(a, b)
  integer :: m = 10
  integer :: i, a(m), b(m)

  !dir$ unroll
  do i = 1, m
    b(i) = a(i) + 1
  end do
end subroutine func1

Example 2:
subroutine func2(m, a, b)
  integer :: i, m, a(m), b(m)

  !dir$ unroll(4)
  do i = 1, m
    b(i) = a(i) + 1
  end do
end subroutine func2
```

4.1.4 NOUNROLL

This directive disables unroll of the loop before which it has been used and is the opposite of UNROLL.

!DIR$ NOUNROLL

Example:

```
subroutine func1(a, b)
  integer :: m = 10
  integer :: i, a(m), b(m)

  !dir$ nounroll
  do i = 1, m
    b(i) = a(i) + 1
  end do
end subroutine func1
```

4.1.5 PREFETCH

This directive is used to insert a hint in the code generator to prefetch instruction for memory references, wherever supported. This allows a better performance in the characteristics of the code. For more information, refer LLVM documentation (*https://llvm.org/docs/LangRef.html#llvm-prefetch-intrinsic*).

!$MEM PREFETCH

Constraints: To enable this directive, compiler optimization level should be in -O0 to -O3.

Example:

```
subroutine prefetch_dir(a1, a2)
  integer :: a1(4096)
  integer :: a2(4096)

  do i = 128, (4096 - 128)
    !$mem prefetch a1, a2(i + 256)
    a1(i) = a2(i - 127) + a2(i + 127)
  end do
end subroutine prefetch_dir
```

4.1.6 Vectorization Pragmas

The compiler directives to control the loop vectorizations are:

- !DIR$ VECTOR
- !DIR$ NOVECTOR
- !DIR$ VECTOR ALWAYS

For these pragmas to be applicable, the optimization levels should be in -O1 to -O3.

`-Menable-vectorize-pragmas` introduced in AOCC 3.1 (to control the vectorization pragmas) is deprecated in AOCC 3.2.

4.1.7 FREEFORM/NOFREEFORM

FREEFORM makes the compiler compile the source in free-form format, while NOFREEFORM directs to compile in a fixed-form format. They are applied to the rest of the file in which they are mentioned, but the directive is reverted if the compiler finds the opposite directive in the same file.

Example:

```
!DIR$ FREEFORM
! This is free-form
temp = a; a = b; b = temp ! Swap a and b
write(6,*) 'Swapped a and b values are =',  &
            a,b           ! Print a and b
!DIR$ NOFREEFORM
C---This-is-fixed-form
```

Chapter 5 Command-line Options

5.1 Clang and Flang Options

5.1.1 Target Selection

Following is the list of all the target selection options:

- `-march=<cpu>`

  Use it to specify if Clang must generate code for a specific processor family member and later. For example, if you specify -march=i486, the compiler can generate instructions that are valid on i486 and later processors, but which may not exist on the earlier ones.

- `-march=znver1`

  Use this architecture option for enabling the best code generation and tuning for AMD “Zen”-based x86 architecture. All the x86 AMD “Zen” ISA and associated intrinsic are supported.

- `-march=znver2`

  Use this architecture option for enabling the best code generation and tuning for AMD “Zen2”-based x86 architecture. All x86 AMD “Zen2” ISA and associated intrinsic are supported.

- `-march=znver3`

  Use this architecture option for enabling best code generation and tuning for AMD “Zen3”-based x86 architecture. All x86 AMD “Zen3” ISA and associated intrinsic are supported.

5.1.2 Driver

- **`-mllvm <options>`** (Applicable for both Clang and Flang)

  Need to provide -mllvm, so that, the option can pass through the compiler front end and is applied on the optimizer where this optimization is implemented.

  For example, `-mllvm -enable-strided-vectorization`

- **`-fuse-ld=lld`** (Applicable only for Clang)

  To invoke lld linker from compiler driver as it is the preferred linker.

Note: *For more information on the Clang options, refer Clang Documentation (https://releases.llvm.org/13.0.0/tools/clang/docs/ClangCommandLineReference.html).*

5.2 Flang Options

For a list of compiler options, use the following commands:

```
$flang -help
$flang -help-hidden
```

The Flang compiler supports all the Clang compiler options (*http://clang.llvm.org/docs/CommandGuide/clang.html*) and the following Flang-specific compiler options:

- `-Kieee`

  It is enabled by default from AOCC 2.2.0.

  It instructs the compiler to conform to the IEEE-754 specifications. The compiler will perform floating-point operations in strict conformance with the IEEE 754 standard. Some optimizations are disabled when this option is specified.

- `-no-flang-libs`

  Do not link against Flang libraries.

- `-mp`

  Enable OpenMP and link with OpenMP library *libomp*.

- `-nomp`

  Do not link with OpenMP library *libomp*.

- `-Mbackslash`

  Treat backslash character like a c-style escape character.

- `-Mno-backslash`

  Treat backslash like any other character.

- `-Mbyteswapio`

  Swap byte-order for unformatted input/output.

- `-Mfixed`

  Assume fixed-format source.

- `-Mextend`

  Allow source lines up to 132 characters.

- `-Mfreeform`

  Assume free-format source.

- `-Mpreprocess`

  Run preprocessor for Fortran files.

- `-Mstandard`

  Check standard conformance.

- `-Msave`

  Assume all variables have SAVE attribute.

- `-module`

  Path to module file (-I also works).

- `-Mallocatable=95`

  Select Fortran 95 semantics for assignments to allocatable objects (default).

- `-Mallocatable=03`

  Select Fortran 03 semantics for assignments to allocatable objects.

- `-static-flang-libs`

  Link using static Flang libraries.

- `-M[no]daz`

  Treat denormalized numbers as zero.

- `-M[no]flushz`

  Set SSE to flush-to-zero mode.

- `-Mcache_align`

  Align large objects on cache-line boundaries.

- `-M[no]fprelaxed`

  This option is ignored.

- `-fdefault-integer-8`

  Treat INTEGER and LOGICAL as INTEGER*8 and LOGICAL*8.

- `-fdefault-real-8`

  Treat REAL as REAL*8.

- `-i8`

  Treat INTEGER and LOGICAL as INTEGER*8 and LOGICAL*8.

- `-r8`

  Treat REAL as REAL*8.

- `-fno-fortran-main`

  Do not link in Fortran main.

- `-Mrecursive`

  Allocate local variables on the stack; thus, allowing recursion. SAVEd, data-initialized, or namelist members are always allocated statically, regardless of the setting of this switch.

- `-Hz,1,0x1`

  Helps preserve array index information for array access expressions which get linearized in the compiler frontend. The preserved information is used by the compiler optimization phase in performing optimizations such as loop transformations. It is recommended that anyone using optimizations, such as loop transformations and other optimizations requiring de-linearized index expressions should use the Hz option. This option has no impact on any other aspects of AOCC's Flang frontend.

5.3 Code Generation and Optimization Options

Both Clang and Flang rely on AOCC optimizer and code generator stages to transform the LLVM IR and generate the best code for the target x86 platform.

Following is the list of optimization options categorized by type:

Optimization Level Options

- -O0

  No optimization: this level compiles the fastest and generates the most debuggable code.

- -O1

  Between the levels -O0 and -O2.

- -O2

  Enables most optimizations.

- -O3

  Enables all optimizations, which take longer to perform or may generate larger code (in an attempt to make the program run faster).

  The -O3 level in AOCC has more optimizations than the base LLVM version. These optimizations include improved handling of indirect calls and advanced vectorization.

- -Ofast

  Enables all the optimizations from -O3 along with other aggressive optimizations that may violate strict compliance with language standards.

  The -Ofast level in AOCC has more optimizations than the base LLVM version. These optimizations include partial unswitching, improvements to inlining, and unrolling.

- -Os

  Similar to the level -O2, but with extra optimizations to reduce the code size.

- -Oz

  Similar to the level -Os (and thus, -O2), but reduces the code size further.

- -O

  Equivalent to the level -O2.

- -O4 and higher

  Equivalent to the level -O3.

Vector Optimization Options

- **`-enable-strided-vectorization`**

  Enables strided memory vectorization as an enhancement to the interleaved vectorization framework present in LLVM. It enables the effective use of gather/scatter instruction patterns. This option must be used along with the interleave vectorization option.

  Usage: `-mllvm -enable-strided-vectorization`

- **`-enable-epilogue-vectorization`**

  Enables vectorization of epilogue-iterations as an enhancement to existing vectorization framework. This enables generation of an additional epilogue vector loop version for the remainder iterations of the original vector loop. The vector size or factor of the original loop should be large enough to allow an effective epilogue vectorization of the remaining iterations. This optimization takes place only when the original vector loop is vectorized with a vector width or factor of sixteen. This vectorization width of sixteen may be overwritten by `-epilogue-vectorization-minimum-VF` command line option.

  Usage: `-mllvm -enable-epilogue-vectorization`

- **`-enable-vectorize-compares`**

  Enables vectorization on certain loops with conditional breaks assuming the memory access are safely bound within the page boundary.

  Usage: `-mllvm -enable-vectorize-compares`

- **`-global-vectorize-slp={true,false}`**

  Vectorizes the straight-line code inside a basic block with data reordering vector operations. This option is set to **true** by default.

  Usage: `-mllvm -global-vectorize-slp={true,false}`

- **`-region-vectorize`**

  Enables vectorization on certain loops with complex control flow which the normal vectorizer cannot handle.

  This optimization is effective with *flto* as the whole program analysis is required to perform this optimization, which can be invoked as `-flto -region-vectorize`.

  Usage: `-mllvm -region-vectorize`

- **`-enable-loop-vectorization-with-conditions`**

  Enables efficient vectorization of loops with conditions by conditionally executing the vector instructions as opposed to flattening the loop body and vectorizing. The vectorized code uses vector versions of compare instructions to guard the instructions in the loop body and uses masked instructions to guard against unsafe memory operations.

  Usage: `-mllvm -enable-loop-vectorization-with-conditions`

- **`-legalize-vector-library-calls`**

  Splits up the unsupported higher vector factor version of vector library calls into supported vector factor version of vector library calls for enabling vectorization.

  Usage: `-mllvm -legalize-vector-library-calls`

- **`-vectorize-noncontigous-memory-aggressively`**

  Enables vectorization involving noncontiguous memory locations by generating multiple loads/ stores and inserts.

Loop Optimization Options

- **`-enable-partial-unswitch`**

  Enables partial loop un-switching, which is an enhancement to the existing loop unswitching optimization in LLVM. Partial loop un-switching hoists a condition inside a loop from a path for which the execution condition remains invariant, whereas the original loop un-switching works for a condition that is completely loop invariant. The condition inside the loop gets hoisted out from the invariant path and original loop is retained for the path where condition is variant.

  Usage: `-mllvm -enable-partial-unswitch`

- **`-aggressive-loop-unswitch`**

  Enables aggressive loop unswitching heuristic (including `-enable-partial-unswitch`) based on the usage of the branch conditional values. Loop unswitching leads to code-bloat. Code-bloat can be minimized if the hoisted condition is executed more often. This heuristic prioritizes the conditions based on the number of times they are used within the loop. The heuristic can be controlled with the following options:

  Usage: `-mllvm-unswitch-identical-branches-min-count=<n>`

  Enables unswitching of a loop with respect to a branch conditional value (B), where B appears in at least **<n>** compares in the loop. This option is enabled with -aggressive-loop-unswitch. The default value is 3.

  ```
  -mllvm -aggressive-loop-unswitch -mllvm -unswitch-identical-branches-min-count=<n>
  ```

  Where, **n** is a positive integer and lower value of **<n>** facilitates more unswitching.

- **`-lv-function-specialization`**

  Generates specialized function versions when the loops inside function are vectorizable and the arguments are not aliased with each other.

  Usage: `-mllvm -lv-function-specialization`

- **`-loop-splitting`**

  Enables splitting of loops into multiple loops to eliminate the branches, which compare the loop induction with an invariant or constant expression. This option is enabled under -O3 by default. To disable this optimization, use `-loop-splitting=false`.

  Usage: `-mllvm -loop-splitting`

- **`-enable-ipo-loop-split`**

  Enables splitting of loops into multiple loops to eliminate the branches, which compares the loop induction with a constant expression. This constant expression can be derived through inter-procedural analysis. This option is enabled under -O3 by default. To disable this optimization, use `-enable-ipo-loop-split=false`.

  Usage: `-mllvm -enable-ipo-loop-split`

- **`-enable-loop-fusion`**

  This option enables the classical loop fusion transformation where the bodies of multiple loop nests are fused into one loop nest. The transformation checks various legality criteria involving the bounds of the loop nests involved, the control flow nesting of the loop nests and so on. The transformation is off by default and may be enabled by the user by using this option. Loop fusion enables reuse of memory access operations across the loop nests and is also beneficial for cache performance. As part of the profitability check for this transformation it uses code size thresholds which control the size of the fused loop body created.

  Usage: `-mllvm -enable-loop-fusion`

- **`-enable-loopinterchange`**

  This option enables the classical loop interchange or loop permutation transformation on a loop nest. It reorders the loops in a multi-dimensional loop nest, checking for various legality criteria in the process. The transformation is off by default and may be enabled by using this option. Loop interchange tries to find a reordering of the loops in a multi-dimensional loop nest such that the number of loop invariant expressions that may be hoisted out from an inner loop to a loop at a higher level may be maximized.

  `-compute-interchange-order`

  This option must be used in combination with the option enable-loopinterchange. It enables the heuristic which determines the best reordering of the loops in a multi-dimensional loop nest such that the number of invariant expressions that may be hoisted out from an inner level loop to an outer one is maximized.

  Usage: `-mllvm -enable-loopinterchange -mllvm -compute-interchange-order`

- **`-fuse-tile-inner-loop`**

  Enables fusion of adjacent tiled loops as a part of loop tiling transformation. This option is set to **false** by default.

  Usage: `-mllvm -fuse-tile-inner-loop`

- **`-enable-loop-distribute-adv`**

  Enables advanced loop distribution which improves loop vectorization by separating out the portions of the loop affecting vectorization. This flag is disabled by default.

  Usage: `-mllvm -enable-loop-distribute-adv`

Math Options

`-convert-pow-exp-to-int={true,false}`

Converts the call to floating point exponent version of pow to its integer exponent version if the floating-point exponent can be converted to integer. This option is set to **true** by default.

Usage: `-mllvm -convert-pow-exp-to-int={true, false}`

Inline Optimization Options

- **`-finline-aggressive`**

  Enables improved inlining capability through better heuristics. This optimization is more effective when using with *flto* as the whole program analysis is required to perform this optimization, which can be invoked as `-flto -finline-aggressive`.

  Usage: `-finline-aggressive`

- **`-inline-recursion=[1,2,3,4]`**

  Enables inlining for recursive functions based on heuristics with level 4 being most aggressive. The default level will be 2. Higher levels may lead to code-bloat due to expansion of recursive functions at call sites.

  – For level 1-2: Enables inlining for recursive functions using heuristics with inline depth 1. Level 2 uses more aggressive heuristics.
  – For level 3: Enables inlining for all recursive functions with inline depth 1.
  – For level 4: Enables inlining for all recursive function with inline depth 10.

  This is more effective with flto as the whole program analysis is required to perform this optimization, which can be invoked as `-mllvm -inline-recursion=[1,2,3,4]`.

  Usage: `-mllvm -inline-recursion=[1,2,3,4]`

Memory Layout Optimization Options

- **`-fstruct-layout`**`=[1,2,3,4,5,6,7]`

  Analyzes the whole program to determine if the structures in the code can be peeled and if the pointer or integer fields in the structure can be compressed. If feasible, this optimization transforms the code to enable these improvements. This transformation is likely to improve cache

utilization and memory bandwidth. It is expected to improve the scalability of programs executed on multiple cores.

This is effective only under *flto* as the whole program analysis is required to perform this optimization. You can choose different levels of aggressiveness with which this optimization can be applied to your application; with 1 being the least aggressive and 7 being the most aggressive level.

- `fstruct-layout=1` enables structure peeling.
- `fstruct-layout=2` enables structure peeling and selectively compresses self-referential pointers in these structures to 32-bit pointers, wherever safe.
- `fstruct-layout=3` enables structure peeling and selectively compresses self-referential pointers in these structures to 16-bit pointers, wherever safe.
- `fstruct-layout=4` enables structure peeling, pointer compression as in level 2 and further enables compression of structure fields, which are of integer type. This is performed under a strict safety check.
- `fstruct-layout=5` enables structure peeling, pointer compression as in level 3 and further enables compression of structure fields which are of integer type. This is performed under a strict safety check.
- `fstruct-layout=6` enables structure peeling, pointer compression as in level 2 and further enables compression of structure fields, which are of type 64-bit **signed int** or **unsigned int**. You must ensure that the values assigned to 64-bit **signed int** fields are in range -(2^31 - 1) to +(2^31 - 1) and 64-bit **unsigned int** fields are in the range 0 to +(2^31 - 1). Else, incorrect results may be obtained. This compression is performed without considering any safety analysis. So, you must ensure the safety based on the program compiled.
- `fstruct-layout=7` enables structure peeling, pointer compression as in level 3 and further enables compression of structure fields, which are of type 64-bit **signed int** or **unsigned int**. You must ensure that the values assigned to 64-bit **signed int** fields are in range -(2^31 - 1) to +(2^31 - 1) and 64-bit **unsigned int** fields are in the range 0 to +(2^31 - 1). Else, incorrect results may be obtained. This compression is performed without considering any safety analysis. So, must ensure the safety based on the program compiled.

Usage: `-fstruct-layout=[1,2,3,4,5,6,7]`

Notes:

1. `fstruct-layout=4` and `fstruct-layout=5` are derived from `fstruct-layout=2` and `fstruct-layout=3` respectively, with the added feature of safe compression of integer fields in structures. Going from `fstruct-layout=4` to `fstruct-layout=5` may result in higher performance if the pointer values are such that the pointers can be compressed to 16-bits.
2. `fstruct-layout=6` and `fstruct-layout=7` are derived from `fstruct-layout=2` and `fstruct-layout=3` respectively, with the added feature of compression of the integer fields in structures. These are similar to `fstruct-layout=4` and `fstruct-layout=5`, but here, the integer fields of the structures are always compressed from 64-bits to 32-bits, without any safety guarantee.

- **`-fremap-arrays`**

  Transforms the data layout of a single dimensional array to provide better cache locality. This optimization is effective only under *flto* as the whole program analysis is required to perform this optimization, which can be invoked as `-flto -fremap-arrays`.

Instruction Level Optimization Options

- **`-enable-X86-prefetching`**

  Enables the generation of x86 prefetch instruction for the memory references inside a loop/ inside an inner most loop of a loop nest to prefetch the second dimension of multidimensional array/ memory references in the inner most of a loop nest.

  Usage: `-mllvm -enable-X86-prefetching`

- **`-suppress-fmas`**

  Identifies the reduction patterns on FMA and suppresses the FMA generation as it is not profitable on the reduction patterns.

  Usage: `-mllvm -suppress-fmas`

- **`-fnt-store`**

  Generates a non-temporal store instruction for array accesses in a loop with a large trip count.

  Usage: `-fnt-store`

- **`-fnt-store=aggressive`**

  Generates non-temporal store instruction for array accesses in a loop, whose iteration count cannot be determined at compile time. In this case, compiler assumes the iteration count is huge.

  Usage: `-fnt-store=aggressive`

- **`-enable-redundant-movs`**

  Removes any redundant mov operations including redundant loads from memory and stores to memory. This can be invoked using `-Wl,-plugin-opt=-enable-redundant-movs`.

  Usage: `-mllvm-enable-redundant-movs`

- **`-merge-constant`**

  Attempts to promote frequently occurring constants to registers. The aim is to reduce the size of the instruction encoding for instructions using constants and obtain a performance improvement.

  Usage: `-mllvm-merge-constant`

Miscellaneous Options

- **`-fitodcalls`**

  Promotes indirect to direct calls by placing conditional calls. Application or benchmarks that have small and deterministic set of target functions for function pointers that are passed as call parameters benefit from this optimization. Indirect-to-direct call promotion transforms the code to use all possible determined targets under runtime checks and falls back to the original code for all

the other cases. Runtime checks are introduced by the compiler for each of these possible function pointer targets followed by direct calls to the targets.

This is a link time optimization, which is invoked as `-flto -fitodcalls`.

- **`-fitodcallsbyclone`**

  Performs value specialization for functions with function pointers passed as an argument. It does this specialization by generating a clone of the function. The cloning of the function happens in the call chain as needed to allow conversion of indirect function call to direct call. This complements `-fitodcalls` optimization and is also a link time optimization, which is invoked as `-flto -fitodcallsbyclone`.

- **`-function-specialize`**

  Optimizes the functions with compile time constant formal arguments.

  Usage: `-mllvm -function-specialize`

- **`-reduce-array-computations=[1,2,3]`**

  Performs array dataflow analysis and optimizes the unused array computations.

  – `reduce-array-computations=1`: Eliminates the computations on unused array elements.
  – `reduce-array-computations=2`: Eliminates the computations on zero valued array elements.
  – `reduce-array-computations=3`: Eliminates the computations on unused and zero valued array elements (combination of 1 and 2).

  This optimization is effective with flto as the whole program analysis is required to perform this optimization, which can be invoked as `-flto -reduce-array-computations=[1,2,3]`.

  Usage: `-mllvm -reduce-array-computations={1,2,3}`

- **`-enable-licm-vrp`**

  Enables estimation of the virtual register pressure before performing loop invariant code motion. This estimation is used to control the number of loop invariants that will be hoisted during the loop invariant code motion.

  Usage: `-mllvm -enable-licm-vrp`

- **`-do-block-reordering={none,normal,aggressive}`**

  Reorders the control predicates in increasing order of complexity from outer predicate to inner. This option is set to normal by default. The **normal** mode reorders simple expressions while the **aggressive** mode reorders the predicates involving function calls despite the presence of code dealing with exceptions.

  This optimization also includes safety analysis which checks if it is safe to reorder the basic blocks. However, when this optimization takes effect, the safety analysis ignores exceptions specifically within the call chain involved in the context of the blocks being reordered. This

would be acceptable for most input programs, but if your program strictly needs to support throwing of exceptions at all points in the program, it is advisable to avoid using this option.

Usage: `-mllvm -do-block-reordering={none, normal, aggressive}`

- **`-favoid-fpe-causing-opt`**

  Restricts a few optimizations that leads to floating point exceptions.

Diagnostics Option(s)

- **`-fsanitize=address|thread|memory|safe-stack`**

  Runs the sanitizers for diagnostics.

Chapter 6 Debuggability

6.1 OpenMP Debugging Support (OMPD)

Note: *This is available in AOCC 2.3 or later.*

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.*

6.2 OMPD Commands

The following table describes the OMPD commands:

Table 2. OMPD Commands

Command	Description
`help ompd`	It lists the subcommands available for OpenMP specific debugging.

Table 2. OMPD Commands

Command	Description
`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.

6.3 OMPD Subcommands

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

Table 3. 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.

6.4 Deprecated Options

The following options have been deprecated:

- `-vectorize-memory-aggressively` (from AOCC 2.2.0)
- `-Menable-vectorize-pragmas=<value>` (from AOCC 3.2.0)

Chapter 7 Support

For support options, the latest documentation, and downloads refer AMD Developer Central (*https://developer.amd.com/amd-aocc/*).

Chapter 8 References

The following document has been used as a reference for this document:

LLVM Documentation (*https://releases.llvm.org/13.0.0/tools/clang/docs/ClangCommandLineReference.html#actions*)