

# AMD

## AOCL User Guide

Publication # **57404**

Revision # **4.1**

Issue Date **August 2023**

## **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.

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

---

<b>Revision History</b>	<b>11</b>
<b>Chapter 1 Introduction</b>	<b>12</b>
1.1 Feature Support Matrix	13
<b>Chapter 2 Supported OS and Compilers</b>	<b>16</b>
2.1 Operating Systems	16
2.2 Compilers	16
2.3 Library	16
2.4 Message Passing Interface (MPI)	16
2.5 Programming Language	16
2.6 Build Utilities	17
<b>Chapter 3 Installing AOCL</b>	<b>18</b>
3.1 Building from Source	18
3.2 Installing AOCL Binary Packages	18
3.2.1 Using Master Package	18
3.2.2 Using Library Package	20
3.2.3 Using Debian and RPM Packages	20
3.2.4 Using Windows Packages	22
<b>Chapter 4 AOCL-BLAS</b>	<b>24</b>
4.1 Installation on Linux	24
4.1.1 Build AOCL-BLAS from Source	24
4.1.2 Using Pre-built Binaries	26
4.2 Application Development Using AOCL-BLAS	26
4.2.1 API Compatibility Layers (Calling AOCL-BLAS)	26
4.2.2 API Compatibility - Advance Options	28
4.2.3 Linking Application with AOCL-BLAS	28
4.2.4 AOCL-BLAS Usage in Fortran	29
4.2.5 AOCL-BLAS Usage in C	31
4.3 Migrating/Porting	35
4.4 Using AOCL-BLAS Library Features	36

4.4.1	Dynamic Dispatch .....	36
4.4.2	AOCL-BLAS - Running the Test Suite .....	39
4.4.3	Testing/Benchmarking .....	40
4.4.4	AOCL-BLAS Utility APIs .....	42
4.5	Debugging and Troubleshooting .....	42
4.5.1	Debugging Build Using GDB .....	42
4.5.2	Viewing Logs .....	43
4.5.3	Checking AOCL-BLAS Operation Progress .....	47
4.6	Build AOCL-BLAS from Source on Windows .....	49
4.6.1	Building AOCL-BLAS using GUI .....	50
4.6.2	Building AOCL-BLAS using Command-line Arguments .....	53
4.6.3	Building and Running the Test Suite .....	54
4.7	LPGEMM in AOCL-BLAS .....	55
4.7.1	Add-on in AOCL-BLAS .....	55
4.7.2	API Naming and Arguments .....	55
4.7.3	Post-operations .....	55
4.7.4	Supported APIs in aocl_gemm .....	56
4.7.5	Enabling aocl_gemm Add-on .....	57
4.7.6	Sample Application 1 .....	57
4.7.7	Sample Application 2 .....	59
<b>Chapter 5</b>	<b>AOCL-LAPACK .....</b>	<b>62</b>
5.1	Installing on Linux .....	62
5.1.1	Building AOCL-LAPACK from Source .....	62
5.1.2	Using Pre-built Libraries .....	65
5.2	Usage on Linux .....	66
5.2.1	Use by Applications .....	66
5.3	Building AOCL-LAPACK from Source on Windows .....	67
5.3.1	Building AOCL-LAPACK Using GUI .....	67
5.3.2	Building AOCL-LAPACK using Command-line Arguments .....	70
5.3.3	Building and Running Test Suite .....	71
5.4	Checking AOCL-LAPACK Operation Progress .....	71

<b>Chapter 6</b>	<b>AOCL-FFTW</b>	<b>74</b>
6.1	Installing	74
6.1.1	Building AOCL-FFTW from Source on Linux	74
6.1.2	Building AOCL-FFTW from Source on Windows	76
6.1.3	Using Pre-built Libraries	80
6.2	Usage	80
6.2.1	Sample Programs for Single-threaded and Multi-threaded FFTW	80
6.2.2	Sample Programs for MPI FFTW	81
6.2.3	Additional Options	81
<b>Chapter 7</b>	<b>AOCL-LibM</b>	<b>82</b>
7.1	Library Contents	82
7.2	Installation	85
7.2.1	Installing the Pre-Built Binaries on Linux	85
7.2.2	Building AOCL-LibM on Linux	86
7.2.3	Building AOCL-LibM on Windows	87
7.3	Using AOCL-LibM	88
<b>Chapter 8</b>	<b>AOCL-ScaLAPACK</b>	<b>90</b>
8.1	Installation	90
8.1.1	Building AOCL-ScaLAPACK from Source on Linux	90
8.1.2	Using Pre-built Libraries	93
8.2	Usage	93
8.3	Building AOCL-ScaLAPACK from Source on Windows	94
8.3.1	Building AOCL-ScaLAPACK Using GUI	94
8.3.2	Building AOCL-ScaLAPACK using Command-line Arguments	97
8.3.3	Building and Running the Individual Tests	97
8.4	Checking AOCL-ScaLAPACK Operation Progress	98
8.5	Additional Features	99
<b>Chapter 9</b>	<b>AOCL-RNG</b>	<b>100</b>
9.1	Installation	100
9.2	Using AOCL-RNG Library on Linux	101
9.3	Using AOCL-RNG Library on Windows	101

<b>Chapter 10</b>	<b>AOCL-SecureRNG</b>	<b>102</b>
10.1	Installation	102
10.2	Usage	102
10.3	Using AOCL-SecureRNG Library on Windows	103
<b>Chapter 11</b>	<b>AOCL-Sparse</b>	<b>105</b>
11.1	Installation	107
11.1.1	Building AOCL-Sparse from Source on Linux	107
11.1.2	Simple Test	109
11.1.3	Using Pre-built Libraries	110
11.2	Building AOCL-Sparse on Linux	110
11.2.1	Use by Applications	110
11.3	Building AOCL-Sparse on Windows	113
11.3.1	Building AOCL-Sparse Using GUI	114
11.3.2	Building AOCL-Sparse using Command-line Arguments	115
11.4	Running an Individual AOCL-Sparse Test	117
11.4.1	Run the Test on Linux	117
11.4.2	Run the test on Windows	118
<b>Chapter 12</b>	<b>AOCL-LibMem</b>	<b>119</b>
12.1	Building AOCL-LibMem for Linux	119
12.2	Running an Application	121
12.3	Running an Application with Tunables	121
12.3.1	Default State	122
12.3.2	Tuned State	122
<b>Chapter 13</b>	<b>AOCL-Cryptography</b>	<b>125</b>
13.1	Requirements	126
13.2	Installation	126
13.2.1	Building AOCL-Cryptography from Source on Linux	126
13.2.2	Building AOCL-Cryptography from Source on Windows	127
13.3	Using AOCL-Cryptography in a Sample Application	129
13.3.1	Compiling and Running Examples	129
13.3.2	AOCL-Cryptography Library Provider for OpenSSL	129

13.3.3	Integrating AOCL Libraries with Applications that Use IPP .....	129
<b>Chapter 14</b>	<b>AOCL-Compression .....</b>	<b>130</b>
14.1	Installation .....	130
14.1.1	Using Pre-built Libraries .....	130
14.1.2	Building from Source .....	131
14.2	Running AOCL-Compression Test Bench on Linux .....	131
14.3	Running AOCL-Compression Test Bench on Windows .....	132
14.4	API Reference .....	133
14.4.1	Unified Standardized API Set .....	133
14.4.2	Interface Data Structures .....	133
14.4.3	Library Return Error Codes .....	135
14.4.4	Native APIs .....	135
14.4.5	Example Test Program .....	138
14.5	Optional Optimization Options .....	139
<b>Chapter 15</b>	<b>AOCL-Utils .....</b>	<b>141</b>
15.1	Requirements .....	141
15.2	Clone and Build the AOCL-Utils Library .....	142
15.3	Using AOCL-Utils .....	142
15.3.1	C API Example .....	142
15.3.2	C++ API Example .....	143
15.3.3	Building on Windows .....	143
15.3.4	Building on Linux .....	144
15.3.5	Output .....	145
<b>Chapter 16</b>	<b>Linking AOCL to Applications .....</b>	<b>146</b>
16.1	High-performance LINPACK Benchmark (HPL) .....	146
16.1.1	Configuring HPL.dat .....	146
16.1.2	Running the Benchmark .....	147
16.2	MUMPS Sparse Solver Library .....	148
16.2.1	Enabling AOCL with MUMPS .....	149
<b>Chapter 17</b>	<b>AOCL Tuning Guidelines .....</b>	<b>154</b>
17.1	AOCL-BLAS Thread Control .....	154

17.1.1	AOCL-BLAS Initialization .....	154
17.1.2	Runtime .....	155
17.2	AOCL Dynamic .....	157
17.2.1	Limitations .....	158
17.3	AOCL-BLAS DGEMM Multi-thread Tuning .....	158
17.3.1	Library Usage Scenarios .....	158
17.3.2	Architecture Specific Tuning .....	159
17.4	AOCL-BLAS DGEMM Block-size Tuning .....	160
17.5	Performance Suggestions for Skinny Matrices .....	162
17.6	AOCL-LAPACK Multi-threading .....	162
17.7	AOCL-FFTW Tuning Guidelines .....	162
<b>Chapter 18</b>	<b>Support .....</b>	<b>165</b>
<b>Chapter 19</b>	<b>References .....</b>	<b>166</b>
<b>Appendix</b>	<b>.....</b>	<b>167</b>
	Check AMD Server Processor Architecture .....	167
	On Linux .....	167
	On Windows .....	167
	Application Notes .....	168
	AOCL-BLAS.....	168
	AOCL-FFTW.....	168



## List of Tables

---

Table 1.	AOCL Feature Support Matrix - 1 .....	13
Table 2.	AOCL Feature Support Matrix - 2 .....	14
Table 3.	AOCL-BLAS API Compatibility Layers .....	27
Table 4.	AOCL-BLAS API Compatibility - Advance Options .....	28
Table 5.	AOCL-BLAS Application - Link Options .....	28
Table 6.	Porting to AOCL-BLAS .....	36
Table 7.	AOCL-BLAS Utility APIs .....	42
Table 8.	Callback Parameters .....	48
Table 9.	CMake Config Options .....	51
Table 10.	GEMM APIs and Supported Post-ops .....	56
Table 11.	Utility APIs in aocl_gemm Add-on .....	56
Table 12.	AOCL-LAPACK Config Options .....	67
Table 13.	AOCL-LAPACK Progress Feature Callback Function Parameters .....	72
Table 14.	AOCL-FFTW Config Options .....	77
Table 15.	Compiler and Type of Library .....	91
Table 16.	AOCL-ScaLAPACK CMake Parameter List .....	95
Table 17.	AOCL-ScaLAPACK Progress Feature Callback Function Parameters .....	98
Table 18.	Additional Features .....	99
Table 19.	Compiler and Library Type .....	108
Table 20.	AOCL-Sparse - CMake Build Options .....	108
Table 21.	Application Implementations .....	122
Table 22.	Sample Threshold Settings .....	124
Table 23.	AOCL-Cryptography - Linux Options .....	126
Table 24.	AOCL-Cryptography - Windows Options .....	128
Table 25.	Optional Optimization Options .....	139
Table 26.	Sample Scenarios - 1 .....	155
Table 27.	Sample Scenarios - 2 .....	156
Table 28.	AOCL Dynamic .....	157

## List of Figures

---

Figure 1.	Sample Run of Function Call Tracing .....	45
Figure 2.	Sample Run with Debug Logs Enabled .....	46
Figure 3.	Debug Logs Showing Input Values of GEMM .....	47
Figure 4.	Microsoft Visual Studio Prerequisites .....	49
Figure 5.	CMake Source and Build Folders.....	50
Figure 6.	Set Generator and Compiler .....	51
Figure 7.	CMake Configure and Generate Project Settings.....	53
Figure 8.	AOCL-LAPACK CMake Configurations .....	70
Figure 9.	AOCL-FFTW CMake Config Options.....	79
Figure 10.	AOCL-ScaLAPACK CMake Options .....	96
Figure 11.	AOCL-ScaLAPACK CMake Config Options .....	96
Figure 13.	AOCL-Sparse CMake Config Options.....	115

## Revision History

---

Date	Revision	Description
August 2023	4.1	<ul style="list-style-type: none"><li>Added Chapter 15</li><li>Added sections 4.7, 8.5, 11.4, 14.1.1, and 14.1.2</li></ul>
November 2022	4.0	<ul style="list-style-type: none"><li>Added sections 9.3, 10.3, 17.1.2.1, and 17.6</li><li>Updated section 4.4.1.3</li><li>Added Chapter 14</li><li>Removed the chapter AOCL-Spack recipes</li></ul>
July 2022	3.2	<ul style="list-style-type: none"><li>Added chapters 12 and 13, sections 5.4, 8.4, and 16.1</li><li>Added Multi-thread support information in chapter 11</li></ul>
December 2021	3.1	Initial version

# Chapter 1 Introduction

---

AMD Optimizing CPU Libraries (AOCL) are a set of numerical libraries optimized for AMD “Zen”-based processors, including EPYC™, Ryzen™ Threadripper™, and Ryzen™. This document provides instructions on installing and using all the AMD optimized libraries.

AOCL is comprised of the following libraries:

- **AOCL-BLAS** is a portable software framework for performing high-performance Basic Linear Algebra Subprograms (BLAS) functionality.
- **AOCL-LAPACK** is a portable library for dense matrix computations that provides the functionality present in the Linear Algebra Package (LAPACK).
- **AOCL-FFTW (Fastest Fourier Transform in the West)** is a comprehensive collection of fast C routines for computing the Discrete Fourier Transform (DFT) and various special cases.
- **AOCL-LibM (AMD Math Library)** is a software library containing a collection of basic math functions optimized for x86-64 processor based machines.
- **AOCL-Utills** is a library which provides APIs to check the available CPU features/flags, cache topology, and so on of AMD “Zen”-based CPUs.
- **AOCL-ScaLAPACK** is a library of high-performance linear algebra routines for parallel distributed memory machines. It depends on external libraries including BLAS and LAPACK for linear algebra computations.
- **AOCL-RNG (AMD Random Number Generator)** is a pseudo-random number generator library.
- **AOCL-SecureRNG** is a library that provides APIs to access the cryptographically secure random numbers generated by the AMD hardware random number generator.
- **AOCL-Sparse** is a library containing the basic linear algebra subroutines for sparse matrices and vectors optimized for AMD “Zen”-based CPUs.
- **AOCL-LibMem** is AMD’s optimized implementation of memory manipulation functions for AMD “Zen”-based CPUs.
- **AOCL-Cryptography** is AMD’s optimized implementation of cryptographic functions.
- **AOCL-Compression** is a software framework of various lossless data compression and decompression methods tuned and optimized for AMD “Zen”-based CPUs.

All the above libraries are open-source except AOCL-RNG.

## 1.1 Feature Support Matrix

Following tables summarize the list of supported features and dependencies for the AOCL libraries:

**Table 1. AOCL Feature Support Matrix - 1**

Library\Feature	AVX512	Dynamic Dispatcher	Vector	Precision
<b>AOCL-BLAS</b>	Yes	Yes	Yes	Single, Double, Complex, Double Complex, Mixed Precision
<b>AOCL-LAPACK</b>	No	Partially (requires AVX2 support)	Not applicable	Single, Double, Complex, Double Complex
<b>AOCL-FFTW</b>	Yes	Yes for Linux with GCC and AOCC. No for Windows with Clang. MSVC compiler has not been used on Windows.	Yes	Single, Double, Long-double, Quad
<b>AOCL-LibM</b>	Yes	Yes	Yes	Single, Double, Complex, Double Complex
<b>AOCL-Sparse</b>	Partial (for SpMV)	Partial (for SpMV)	Yes	Single, Double
<b>AOCL-Cryptography</b>	Yes	Yes, GCC and AOCC on Linux; Clang on Windows.	Not applicable	Not applicable
<b>AOCL-Compression</b>	AVX512 instructions have not been used. But, library can be built with -mavx512f compiler option.	Yes, GCC and AOCC on Linux; Clang on Windows.	Yes	Not applicable
<b>AOCL-RNG</b>	Partial	Yes	Not applicable	Single, Double
<b>AOCL-SecureRNG</b>	Not applicable	Not applicable	Not applicable	Not applicable
<b>AOCL-ScaLAPACK</b>	Dependent on the underlying BLAS and LAPACK libraries	Dependent on the underlying BLAS and LAPACK libraries	Not applicable	Single, Double, Complex, Double Complex

**Table 1. AOCL Feature Support Matrix - 1**

Library\Feature	AVX512	Dynamic Dispatcher	Vector	Precision
AOCL-LibMem	Yes	No	Yes	Not applicable
AOCL-Utills	Not applicable	Not applicable	Not applicable	Not applicable

**Table 2. AOCL Feature Support Matrix - 2**

Library\Feature	glibc Dependency	Single-threaded	Multi-threaded	MPI
AOCL-BLAS	Yes	Yes	Yes	No
AOCL-LAPACK	Yes	Yes	Yes	No
AOCL-FFTW	Yes	Yes	Yes	Yes
AOCL-LibM	Yes	Yes	No	No
AOCL-Sparse	Yes	Yes	Partial (for SpMV)	No
AOCL-Cryptography	Yes	Yes	No	No
AOCL-Compression	Yes	Yes	No	No
AOCL-RNG	Yes	Yes	No	No
AOCL-SecureRNG	No	Yes	No	No
AOCL-ScaLAPACK	Yes	Yes, dependent on the underlying BLAS and LAPACK libraries	Yes, dependent on the underlying BLAS and LAPACK libraries	Yes
AOCL-LibMem	Yes	Yes	No	No
AOCL-Utills	Yes	Yes	No	No

Dynamic Dispatch facilitates building a single binary compatible with all the AMD “Zen” architectures. At the runtime, this feature enables optimizations specific to the detected AMD “Zen” architecture.

You can find the flags to enable/disable (the applicable features in [Table 1](#) and [Table 2](#)) in the individual library sections.

Additionally, AMD provides Spack (<https://spack.io/>) recipes for installing AOCL-BLAS, AOCL-LAPACK, AOCL-ScaLAPACK, AOCL-LibM, AOCL-FFTW, AOCL-Sparse, and AOCL-Utils libraries.

For more information on the AOCL release and installers, refer the AMD Developer Central (<https://www.amd.com/en/developer/aocl.html>).

For any issues or queries on the libraries, send an email to [toolchainsupport@amd.com](mailto:toolchainsupport@amd.com).

To determine the underlying architecture of your AMD system, refer to Check AMD Server Processor Architecture.

---

## Chapter 2 Supported OS and Compilers

---

This section lists the supported operating systems, compilers, and prerequisites for AOCL 4.1. This release has been validated on the following:

***Note:** For the supported compiler versions and prerequisites of a specific library, refer to the corresponding sections.*

### 2.1 Operating Systems

- Ubuntu<sup>®</sup> 20.04 LTS and 22.04 LTS
- Red Hat<sup>®</sup> Enterprise Linux<sup>®</sup> (RHEL) 9.0 and 8.6
- SUSE Linux Enterprise Server (SLES) 15 SP3
- Windows Server 2019
- Windows<sup>®</sup> 10
- Windows 11 Pro

### 2.2 Compilers

- GCC 12.2 and 13.1
- AOCC 3.2, 4.0, and 4.1
- LLVM<sup>™</sup> 15 and 16

### 2.3 Library

- glibc 2.28 and 2.35
- OpenSSL 3.0.0 through 3.0.7

### 2.4 Message Passing Interface (MPI)

Open MPI 4.1.4

### 2.5 Programming Language

- Python versions 3.4, 3.6, 3.8, and 3.9
- Perl 5.14 and 5.34



## 2.6 Build Utilities

- GNU Make 4.3
- CMake 3.20.2, 3.22.1, and 3.26.2
- Microsoft Visual Studio 2019 (build 16.8.7)/2022 (build 17.3.2)

---

## Chapter 3 Installing AOCL

---

### 3.1 Building from Source

You can download the following open-source libraries of AOCL from GitHub and build from source:

- AOCL-BLAS (<https://github.com/amd/blis>)
- AOCL-LAPACK (<https://github.com/amd/libflame>)
- AOCL-FFTW (<https://github.com/amd/amd-fftw>)
- AOCL-LibM (<https://github.com/amd/aocl-libm-ose>)
- AOCL-ScaLAPACK (<https://github.com/amd/aocl-scalapack>)
- AOCL-Sparse (<https://github.com/amd/aocl-sparse>)
- AOCL-Cryptography (<https://github.com/amd/aocl-crypto>)
- AOCL-Compression (<https://github.com/amd/aocl-compression>)
- AOCL-LibMem (<https://github.com/amd/aocl-libmem>)
- AOCL-Utills (<https://github.com/amd/aocl-utils>)

The details on installing from source for each library is explained in the later sections. For more information on Spack-based installation of AOCL libraries, refer to AMD Developer Central (<https://www.amd.com/en/developer/spack/spac-aocl.html>).

### 3.2 Installing AOCL Binary Packages

The section describes the procedure to install AOCL binaries on Linux and Windows.

#### 3.2.1 Using Master Package

Complete the following steps to install the AOCL library suite:

1. Download the AOCL tar packages from the **Download** (<https://www.amd.com/en/developer/aocl.html#downloads>) section to the target machine.
2. Use the command `tar -xvf <aocl-linux-<compiler>-4.1.0.tar.gz>` to untar the package.

The installer file `install.sh` is available in `aocl-linux-<compiler>-4.1.0`.

- Run `./install.sh` to install the AOCL package (all libraries) to the default `INSTALL_PATH`: `/home/<username>/aocl/4.1.0/<compiler>`, where the compiler value is `aocc` or `gcc`.

Use `install.sh` to print the usage of the script. A few supported options are:

- h — Print the help.
  - t — Custom target directory to install libraries.
  - l — Library to be installed.
  - i — Select LP64/ILP64 libraries to be set as default.
- To install the AOCL package in a custom location, use the installer with the option: `-t <CUSTOM_PATH>`. For example, `./install.sh -t /home/<username>`.
  - You can use the master installer to install the individual library out of the master package. The library names used are `blis`, `libflame`, `libm`, `scalapack`, `rng`, `secrng`, `fftw`, `compression`, `crypto`, and `sparse`. You can do one of the following:
    - To install a specific library, use the option: `-l <Library name>`. For example, `./install.sh -l blis`.
    - Install the individual library in a path of your choice. For example, `./install.sh -t /home/amd -l libm`.
  - AOCL libraries support the following two integer types:
    - LP64 libraries and header files are installed in `/INSTALL_PATH/lib_LP64` and `/INSTALL_PATH/include_LP64` respectively.
    - ILP64 libraries and header files are installed in `/INSTALL_PATH/lib_ILP64` and `/INSTALL_PATH/include_ILP64` respectively.

**Note:** *AOCL-Compression supports only LP64; AOCL-FFTW supports LP64 and ILP64 in single binary using a different set of APIs.*

By default, LP64 libraries and header files are available in `/INSTALL_PATH/lib` and `/INSTALL_PATH/include` respectively.

Suffix `./install.sh` with `-i <lp64/ilp64>` to:

- Set the LP64 libraries as the default libraries, use the installer with the option: `-i lp64`. For example, `./install.sh -t /home/amd -l blis -i lp64`.

This installs only AOCL-BLAS library in the path `/home/amd` and sets LP64 AOCL-BLAS libraries as the default.

- Set ILP64 libraries as the default use the installer with the option: `-i ilp64`. For example, `./install.sh -i ilp64`.

This installs all AOCL libraries in the default path and sets ILP64 libraries as the default.

### 3.2.2 Using Library Package

Refer to the AOCL home page (<https://www.amd.com/en/developer/aocl.html#downloads>) to download the individual library binaries from the respective pages.

For example, AOCL-BLAS and AOCL-LAPACK tar packages are available in the BLAS library page (<https://www.amd.com/en/developer/aocl/blas.html>).

### 3.2.3 Using Debian and RPM Packages

The Debian and RPM packages of AOCL are available in the **Download** section (<https://www.amd.com/en/developer/aocl.html#downloads>).

The package name used in the following installation procedure is based on the ‘gcc’ build. For the AOCC build, you can replace ‘gcc’ with ‘aocc’.

#### Installing Debian Package

Complete the following steps to install the AOCL Debian package:

1. Download the AOCL 4.1 Debian package to the target machine.
2. Check the installation path before installing.

```
$ dpkg -c aocl-linux-gcc-4.1.0_1_amd64.deb
```

3. Install the package.

```
$ sudo dpkg -i aocl-linux-gcc-4.1.0_1_amd64.deb
Or
$ sudo apt install ./aocl-linux-gcc-4.1.0_1_amd64.deb
```

**Note:** You must have the *sudo* privileges to perform this action.

4. Display the installed package information along with the package version and a short description.

```
$ dpkg -s aocl-linux-gcc-4.1.0
```

5. List the contents of the package.

```
$ dpkg -L aocl-linux-gcc-4.1.0
```

6. AOCL libraries support the following two integer types:

- LP64 libraries and header files are installed in */INSTALL\_PATH/lib\_LP64* and */INSTALL\_PATH/include\_LP64* respectively.
- ILP64 libraries and header files are installed in */INSTALL\_PATH/lib\_ILP64* and */INSTALL\_PATH/include\_ILP64* respectively.

**Note:** *AOCL-Compression supports only LP64; AOCL-FFTW supports LP64 and ILP64 in single binary using a different set of APIs.*

By default, LP64 libraries and header files are available in `/INSTALL_PATH/lib` and `/INSTALL_PATH/include` respectively, where:

- `INSTALL_PATH`: `/opt/AMD/aocl/aocl-linux-<compiler>-4.1.0/<compiler>`
- Compiler: `aocc` or `gcc`

For example, `INSTALL_PATH` for `aocc` compiler is:

`/opt/AMD/aocl/aocl-linux-aocc-4.1.0/aocc`

7. To change the default library path to ILP64 / LP64, use the script as follows:

```
cd /opt/AMD/aocl/aocl-linux-<compiler>-4.1.0/aocc
sudo bash set_aocl_interface_symlink.sh <ilp64 / lp64>
```

## Uninstalling Debian package

Execute one of the following commands to uninstall the AOCL Debian package:

```
$ sudo dpkg -r aocl-linux-gcc-4.1.0
or
$ sudo apt remove aocl-linux-gcc-4.1.0
```

## Installing RPM Package

Complete the following steps to install the AOCL RPM package:

1. Download the AOCL 4.1 RPM package to the target machine.
2. Install the package.

```
$ sudo rpm -ivh aocl-linux-gcc-4.1.0-1.x86_64.rpm
```

**Note:** *You must have the sudo privileges to perform this action.*

3. Display the installed package information along with the package version and a short description.

```
$ rpm -qi aocl-linux-gcc-4.1.0.x86_64
```

4. List the contents of the package.

```
$ rpm -ql aocl-linux-gcc-4.1.0
```

5. AOCL libraries support the following two integer types:

- LP64 libraries and header files are installed in `/INSTALL_PATH/lib_LP64` and `/INSTALL_PATH/include_LP64` respectively.
- ILP64 libraries and header files are installed in `/INSTALL_PATH/lib_ILP64` and `/INSTALL_PATH/include_ILP64` respectively.

**Note:** AOCL-Compression supports only LP64; AOCL-FFTW supports LP64 and ILP64 in single binary using a different set of APIs.

By default, LP64 libraries and header files are available in `/INSTALL_PATH/lib` and `/INSTALL_PATH/include` respectively.

Where,

- `INSTALL_PATH`: `/opt/AMD/aocl/aocl-linux-<compiler>-4.1.0/<compiler>`
- Compiler: aocc or gcc

For example, `INSTALL_PATH` for aocc compiler is:

`/opt/AMD/aocl/aocl-linux-aocc-4.1.0/aocc`

6. To change the default library path to ILP64 / LP64, use the script as follows:

```
cd /opt/AMD/aocl/aocl-linux-<compiler>-4.1.0/aocc
sudo bash set_aocl_interface_symlink.sh <ilp64 / lp64>
```

### Uninstalling RPM package

Execute the following command to uninstall the AOCL RPM package:

```
$ rpm -e aocl-linux-gcc-4.1.0
```

## 3.2.4 Using Windows Packages

### Installing a Windows Package

Complete the following steps to install the AOCL Windows package:

1. Download the AOCL Windows installer from the Download (<https://www.amd.com/en/developer/aocl.html#downloads>) section.
2. Double-click the executable.  
The installation wizard is displayed.
3. Click the **Next** button.
4. Accept the **License Agreement** and click the **Next** button.
5. Select the libraries to be installed and the destination folder.
6. Click the **Install** button to begin the installation.
7. Click the **Finish** button to complete the installation.

## Uninstalling a Windows Package

Complete the following steps to uninstall the AOCL Windows binaries:

1. Double-click the AOCL Windows installer.
2. Click the **Remove** button.

Alternatively, you can also use the **Add or remove programs** option in Windows.

3. Click the **Finish** button to complete the uninstallation.

## Chapter 4 AOCL-BLAS

---

AOCL-BLAS is a high-performant implementation of the Basic Linear Algebra Subprograms (BLAS). The BLAS was designed to provide the essential kernels of matrix and vector computation and are the most commonly used computationally intensive operations in dense numerical linear algebra. Select kernels have been optimized for the AMD “Zen”-based processors, for example, AMD EPYC™, AMD Ryzen™, AMD Ryzen™ Threadripper™ processors by AMD and others.

AOCL\_BLAS is developed as a forked version of BLIS (<https://github.com/flame/blis>), which is developed by members of the Science of High-Performance Computing (SHPC) group in the Institute for Computational Engineering and Sciences at The University of Texas at Austin and other collaborators (including AMD). All known features and functionalities of BLIS are retained and supported in AOCL-BLAS library, along with the standard BLAS and CBLAS interfaces. C++ template interfaces for the BLAS functionalities are also included.

### 4.1 Installation on Linux

You can install AOCL-BLAS from source or pre-built libraries.

#### 4.1.1 Build AOCL-BLAS from Source

GitHub URL: <https://github.com/amd/blis>

You can use the following ways to build AOCL-BLAS using the configure/make method:

- **auto** — This configuration generates a binary optimized for the build machine’s AMD “Zen” core architecture. This is useful when you build the library on the target system. Starting from the AOCL-BLAS 2.1 release, the **auto** configuration option enables selecting the appropriate build configuration based on the target CPU architecture. For example, for a build machine using the 1<sup>st</sup> Gen AMD EPYC™ (code name "Naples") processor, the **zen** configuration will be auto-selected. For a build machine using the 2<sup>nd</sup> Gen AMD EPYC™ processor (code name "Rome"), the **zen2** configuration will be auto-selected. From AOCL-BLAS 3.0 forward, **zen3** will be auto-selected for the 3<sup>rd</sup> Gen AMD EPYC™ processor (code name "Milan"). From AOCL-BLAS 4.0 forward, **zen4** will be auto-selected for the 4<sup>th</sup> Gen AMD EPYC™ processors (code name "Genoa" or "Bergamo").
- **zen** — This configuration generates a binary compatible with AMD “Zen” architecture and is optimized for it. The architecture of the build machine is not relevant.
- **zen2** — This configuration generates binary compatible with AMD “Zen2” architecture and is optimized for it. The architecture of the build machine is not relevant.
- **zen3** — This configuration generates binary compatible with AMD “Zen3” architecture and is optimized for it. The architecture of the build machine is not relevant.



- **zen4** — This configuration generates binary compatible with AMD “Zen4” architecture and is optimized for it. The architecture of the build machine is not relevant.
- **amdzen** — The library built using this configuration generates a binary compatible with and optimized for AMD “Zen”, AMD “Zen2”, AMD “Zen3”, and AMD “Zen4” architectures. The architecture of the build machine is not relevant. The architecture of the target machine is checked during the runtime, based on which, the relevant optimizations are picked up automatically.

This feature is also called Dynamic Dispatch. For more information, refer “Dynamic Dispatch” on page 36.

Depending on the target system and the build environment, you must enable/disable the appropriate configure options. The following sub-sections provide instructions for compiling AOCL-BLAS. For a complete list of the options and their descriptions, use the command `./configure --help`.

#### 4.1.1.1 Single-thread AOCL-BLAS

Complete the following steps to install a single-thread AOCL-BLAS:

1. Clone the AOCL-BLAS Git repository (<https://github.com/amd/blis.git>).
2. Configure the library as required:

##### **GCC (Default)**

```
$ ./configure --enable-cblas --prefix=<your-install-dir> auto
```

##### **AOCC**

```
$ ./configure --enable-cblas --prefix=<your-install-dir> --complex-return=intel CC=clang CXX=clang++ auto
```

3. To build the library, use the command “`$ make`”.
4. To install the library on build machine, use the command “`$ make install`”.

#### 4.1.1.2 Multi-thread AOCL-BLAS

Complete the following steps to install a multi-thread AOCL-BLAS:

1. Clone the AOCL-BLAS Git repository (<https://github.com/amd/blis.git>).
2. Configure the library as required:

##### **GCC (Default)**

```
$ ./configure --enable-cblas --enable-threading=[Mode] --prefix=<your-install-dir> auto
```

##### **AOCC**

```
$ ./configure --enable-cblas --enable-threading=[Mode] --prefix=<your-install-dir> --complex-return=intel CC=clang CXX=clang++ auto
```

[Mode] values can be openmp and no. “no” will disable multi-threading.

3. To build the library, use the command “`make`”.
4. To install the library on build machine, use the command “`make install`”.

#### 4.1.1.3 Verifying AOCL-BLAS Installation

The AOCL-BLAS source directory contains the test cases which demonstrate the usage of AOCL-BLAS APIs.

To execute the tests, navigate to the AOCL-BLAS source directory and run the following command:

```
$ make check
```

Execute the AOCL-BLAS C++ Template API tests as follows:

```
$ make checkcpp
```

#### 4.1.2 Using Pre-built Binaries

AOCL-BLAS library binaries for Linux are available at the following URL:

<https://www.amd.com/en/developer/aocl/blas.html>

Also, the AOCL-BLAS binary can be installed from the AOCL master installer tar file (<https://www.amd.com/en/developer/aocl.html>).

The master installer includes the following:

- Single threaded and multi-threaded AOCL-BLAS binaries.
- Binaries built with **amdzen** config with LP64 and ILP64 integer support.
- Multi-threaded AOCL-BLAS binary (libblis-mt) built with OpenMP threading mode.

The tar file includes pre-built binaries of other AMD libraries as explained in “Using Master Package” on page 18.

## 4.2 Application Development Using AOCL-BLAS

This section explains the different types of APIs provided by AOCL-BLAS. It describes how to call them and link with the library.

#### 4.2.1 API Compatibility Layers (Calling AOCL-BLAS)

AOCL-BLAS supports various API compatibility layers. The following sub-sections explain these layers with source code examples.

The standard BLAS/CBLAS layers allows portability between various libraries.

AOCL-BLAS also includes BLIS APIs that provide more flexibility and control to help achieve the best performance in some situations.

The following table lists all the supported layers and the configure options to control them, with the default setting in bold:

**Table 3. AOCL-BLAS API Compatibility Layers**

API Compatibility Layer	Header Files	Configuration Option	Usages
BLAS (Fortran)	Not applicable	<b>--enable-blas</b> <b>--disable-blas</b>	Use this option when calling AOCL-BLAS from Fortran applications.  API Name Format: DGEMM
BLAS (C)	blis.h	<b>--enable-blas</b> <b>--disable-blas</b>	Use this option when calling AOCL-BLAS from C application using BLAS type parameters.  API Name Format: dgemm_
CBLAS	cblas.h	<b>--enable-cblas</b> (Implies <b>--enable-blas</b> ) <b>--disable-cblas</b>	Use this option when calling AOCL-BLAS from C application using CBLAS type parameters.  API Name Format: cblas_dgemm
BLIS - C Non-standard	blis.h	Default	This is AOCL-BLAS library specific (non-standard) interface, it provides most flexibility in calling AOCL-BLAS for best performance. However, these applications will not be portable to other BLAS/CBLAS compatible libraries.  API Name Format: bli_gemm API Name Format: blis_gemm_ex
BLIS – CPP Non-standard	blis.hh	Default	This is AOCL-BLAS library specific (non-standard) C++ interface. This interface follows same parameter order as CBLAS. However, these applications will not be portable to other BLAS/CBLAS compatible libraries.  API Name Format: blis::gemm

## 4.2.2 API Compatibility - Advance Options

The API compatibility can be further extended to meet additional requirements for input sizes (ILP64) and different ways in which complex numbers are handled. The following table explains such options:

**Table 4. AOCL-BLAS API Compatibility - Advance Options**

Feature	Configuration Option	Usages
ILP64 Support	<code>--blas-int-size=SIZE</code>	This option can be used to specify the integer types used in external BLAS/CBLAS interfaces.  Accepted Values: ILP64 - SIZE = 64 LP64 - SIZE = 32 (Default)
Complex Number return handling	<code>--complex-return=gnu intel</code>	The complex numbers can be returned through registers or the hidden parameter. Based on the way application is calling the API, the library must be configured to match the return value receptions. gnu = return complex values through registers intel = return complex values through hidden parameter. For more information and example, refer “Returning Complex Numbers” on page 35.

## 4.2.3 Linking Application with AOCL-BLAS

The AOCL-BLAS library can be linked statically or dynamically with the user application. It has a separate binary for single-threaded and multi-threaded implementation.

The basic build command is as following:

```
gcc test_blis.c -I<path-to-AOCL-BLAS-header> <link-options> -o test_blis.x
```

The following table explains different options depending on a particular build configuration:

**Table 5. AOCL-BLAS Application - Link Options**

Application Type	Linking Type	Link Options
Single-threaded	Static	<code>&lt;path-to-AOCL-BLAS-library&gt;/libblis.a -lm -lpthread</code>
Single-threaded	Dynamic	<code>-L&lt;path-to-AOCL-BLAS-library&gt; -lblis -lm -lpthread</code>
Multi-threaded	Static	<code>&lt;path-to-AOCL-BLAS-library&gt;/libblis-mt.a -lm -fopenmp</code>
Multi-threaded	Dynamic	<code>-L&lt;path-to-AOCL-BLAS-library&gt; -lblis-mt -lm -fopenmp</code>

### 4.2.3.1 Example - Dynamic Linking and Execution

AOCL-BLAS can be built as a shared library. By default, the library is built as both static and shared libraries. Complete the following steps to build a shared lib version of AOCL-BLAS and link it with the user application:

1. During configuration, enable the support for the shared lib using the following command:

```
./configure --disable-static --enable-shared zen
```

2. Link the application with the generated shared library using the following command:

```
gcc CBLAS_DGEMM_usage.c -I path/to/include/aocl-blas/ -L path/to/libblis.so -lblis -lm -lpthread -o CBLAS_DGEMM_usage.x
```

3. Ensure that the shared library is available in the library load path. Run the application using the following command (for this demo we will use the *BLAS\_DGEMM\_usage.c*):

```
$ export LD_LIBRARY_PATH="path/to/libblis.so"
```

```
$ ./BLAS_DGEMM_usage.x
```

```
a =
1.000000      2.000000
3.000000      4.000000
b =
5.000000      6.000000
7.000000      8.000000
c =
19.000000     22.000000
43.000000     50.000000
```

### 4.2.4 AOCL-BLAS Usage in Fortran

AOCL-BLAS can be used with the Fortran applications through the standard BLAS API.

#### 4.2.4.1 Using BLAS API in Fortran

For example, the following Fortran code does a double precision general matrix-matrix multiplication. It calls the 'DGEMM' BLAS API function to accomplish this. A sample command to compile and link it with the AOCL-BLAS library is shown in the following code:

```
! File: BLAS_DGEMM_usage.f
! Example code to demonstrate BLAS DGEMM usage

program dgemm_usage

implicit none

EXTERNAL DGEMM

DOUBLE PRECISION, ALLOCATABLE :: a(:, :)
DOUBLE PRECISION, ALLOCATABLE :: b(:, :)
DOUBLE PRECISION, ALLOCATABLE :: c(:, :)
INTEGER I, J, M, N, K, lda, ldb, ldc
DOUBLE PRECISION alpha, beta

M=2
N=M
K=M
lda=M
ldb=K
ldc=M
alpha=1.0
beta=0.0

ALLOCATE(a(lda,K), b(ldb,N), c(ldc,N))

a=RESHAPE((/ 1.0, 3.0, &
             2.0, 4.0 /), &
           (/lda,K/))
b=RESHAPE((/ 5.0, 7.0, &
             6.0, 8.0 /), &
           (/ldb,N/))

WRITE(*,*) ("a =")
DO I = LBOUND(a,1), UBOUND(a,1)
  WRITE(*,*) (a(I,J), J=LBOUND(a,2), UBOUND(a,2))
END DO
WRITE(*,*) ("b =")
DO I = LBOUND(b,1), UBOUND(b,1)
  WRITE(*,*) (b(I,J), J=LBOUND(b,2), UBOUND(b,2))
END DO

CALL DGEMM('N', 'N', M, N, K, alpha, a, lda, b, ldb, beta, c, ldc)

WRITE(*,*) ("c =")
DO I = LBOUND(c,1), UBOUND(c,1)
  WRITE(*,*) (c(I,J), J=LBOUND(c,2), UBOUND(c,2))
END DO

end program dgemm_usage
```

A sample compilation command with gfortran compiler for the code above:

```
gfortran -ffree-form BLAS_DGEMM_usage.f path/to/libblis.a
```

## 4.2.5 AOCL-BLAS Usage in C

The AOCL-BLAS library supports standard BLAS, CBLAS, and BLIS APIs. They can be called from C or C++ programs. BLAS and CBLAS examples are available at:

<https://github.com/amd/blis/blob/master/docs/BLISObjectAPI.md>

Details on the BLIS interfaces are available at:

<https://github.com/amd/blis/blob/master/docs/BLISTypedAPI.md>

### 4.2.5.1 Using BLAS API in C

Following is the C version of the Fortran code in section 4.2.4. It uses the standard BLAS API.

The following process takes place during the execution of the code:

1. The matrices are transposed to account for the row-major storage of C and the column-major convention of BLAS (inherited from Fortran).
2. The function arguments are passed by address again to be in line with Fortran conventions.
3. There is a trailing underscore in the function name ('dgemm\_') as BLAS APIs require Fortran compilers to add a trailing underscore.

4. "blis.h" is included as a header. A sample command to compile it and link with the AOCL-BLAS library is also shown in the following code:

```
// File: BLAS_DGEMM_usage.c
// Example code to demonstrate BLAS DGEMM usage

#include<stdio.h>
#include "blis.h"

#define DIM 2

int main() {

double a[DIM * DIM] = { 1.0, 3.0, 2.0, 4.0 };
double b[DIM * DIM] = { 5.0, 7.0, 6.0, 8.0 };
double c[DIM * DIM];
int I, J, M, N, K, lda, ldb, ldc;
double alpha, beta;

M = DIM;
N = M;
K = M;
lda = M;
ldb = K;
ldc = M;
alpha = 1.0;
beta = 0.0;

printf("a = \n");
for ( I = 0; I < M; I ++ ) {
for ( J = 0; J < K; J ++ ) {
printf("%f\t", a[J * K + I]);
}
printf("\n");
}
printf("b = \n");
for ( I = 0; I < K; I ++ ) {
for ( J = 0; J < N; J ++ ) {
printf("%f\t", b[J * N + I]);
}
printf("\n");
}

dgemm_("N","N",&M,&N,&K,&alpha,a,&lda,b,&ldb,&beta,c,&ldc);

printf("c = \n");
for ( I = 0; I < M; I ++ ) {
for ( J = 0; J < N; J ++ ) {
printf("%f\t", c[J * N + I]);
}
printf("\n");
}

return 0;
}
```



A sample compilation command with a gcc compiler for the code above:

```
gcc BLAS_DGEMM_usage.c -Ipath/to/include/aocl-blas/ -lpthread -lm path/to/libblis.a
```

#### 4.2.5.2 Example Application - Using AOCL-BLAS with CBLAS API

This section contains an example application written in C code using the CBLAS API for DGEMM.

The following process takes place during the execution of the code:

1. The CBLAS Layout option is used to choose row-major layout which is consistent with C.
2. The function arguments are passed by value.

3. "cblas.h" is included as a header. A sample command to compile it and link with the AOCL-BLAS library is also shown in the following code:

```
// File: CBLAS_DGEMM_usage.c
// Example code to demonstrate CBLAS DGEMM usage
#include<stdio.h>
#include "cblas.h"

#define DIM 2

int main() {
double a[DIM * DIM] = { 1.0, 2.0, 3.0, 4.0 };
double b[DIM * DIM] = { 5.0, 6.0, 7.0, 8.0 };
double c[DIM * DIM];
int I, J, M, N, K, lda, ldb, ldc;
double alpha, beta;

M = DIM;
N = M;
K = M;
lda = M;
ldb = K;
ldc = M;
alpha = 1.0;
beta = 0.0;

printf("a = \n");
for ( I = 0; I < M; I ++ ) {
for ( J = 0; J < K; J ++ ) {
printf("%f\t", a[I * K + J]);
}
printf("\n");
}
printf("b = \n");
for ( I = 0; I < K; I ++ ) {
for ( J = 0; J < N; J ++ ) {
printf("%f\t", b[I * N + J]);
}
printf("\n");
}

cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, M, N, K, alpha, a, lda, b, ldb, beta,
c, ldc);

printf("c = \n");
for ( I = 0; I < M; I ++ ) {
for ( J = 0; J < N; J ++ ) {
printf("%f\t", c[I * N + J]);
}
printf("\n");
}

return 0;
}
```

**Note:** To get the CBLAS API with AOCL-BLAS, you must provide the flag '--enable-cblas' to the 'configure' command while building the AOCL-BLAS library.

A sample compilation command with a gcc compiler for the code above is as follows:

```
gcc CBLAS_DGEMM_usage.c -Ipath/to/include/aocl-blas/ -lpthread -lm path/to/libblis.a
```

### 4.2.5.3 Returning Complex Numbers

The GNU Fortran compiler (gfortran), AOCC (Flang), and Intel Fortran compiler (ifort) have different requirements for returning complex numbers from the C functions as follows:

- GNU (gfortran)/AOCC (Flang) compiler returns complex numbers using registers. Thus, the complex numbers are returned as the return value of the function itself.
- Intel® (ifort) compiler returns complex numbers using hidden first argument. The caller must pass the pointer to the return value as the first parameter.

#### gfortran Example:

- Configure Option:

```
--complex-return=gnu
```

- API Call:

```
ret_value = cdotc_(&n, x, &incx, y, &incy);
```

#### ifort example:

- Configure Option:

```
--complex-return=intel
```

- API Call:

```
cdotc_(&ret_value, &n, x, &incx, y, &incy);
```

This feature is currently enabled only for cdotx and zdotx APIs.

## 4.3 Migrating/Porting

The application written for MKL, OpenBLAS or any other library using standard BLAS or CBLAS interfaces can be ported to AOCL-BLAS with minimal or no changes.

Complete the following steps to port from BLAS or CBLAS to AOCL-BLAS:

1. Update the source code to include the correct header files.
2. Update the build script or makefile to use correct compile or link option.

The following table lists the compiler and linker options to use while porting to AOCL-BLAS:

**Table 6. Porting to AOCL-BLAS**

	MKL	OpenBLAS	AOCL-BLAS	
			Single-threaded	Multi-threaded
<b>Header File</b>	mkl.h	cblas.h	blis.h/cblas.h	blis.h/cblas.h
<b>Link Options</b>	-lmkl_intel_lp64 -lmkl_core -lmkl_blacs_intelmpi_ilp64 -lmkl_intel_thread	-lopenblas	-lm -lblis - lpthread	-lm -fopenmp -lblis-mt

## 4.4 Using AOCL-BLAS Library Features

### 4.4.1 Dynamic Dispatch

Starting from AOCL 3.1, AOCL-BLAS supports Dynamic Dispatch feature. It enables you to use the same binary with different code paths optimized for different architectures.

#### 4.4.1.1 Purpose

Before Dynamic Dispatch, the user had to build different binaries for each CPU architecture, that is, AMD “Zen”, AMD “Zen2”, and AMD “Zen3” architectures. Furthermore, when building the application, users had to ensure that they used the correct AMD “Zen”-based library as needed for the platform. This becomes challenging when using AOCL-BLAS on a cluster having nodes of different architectures.

Dynamic Dispatch addresses this issue by building a single binary compatible with all the AMD “Zen” architectures. At the runtime, the Dynamic Dispatch feature enables optimizations specific to the detected AMD “Zen” architecture.

#### 4.4.1.2 On non-AMD “Zen” Architectures

The Dynamic Dispatch feature supports AMD “Zen”, AMD “Zen2”, AMD “Zen3”, and AMD “Zen4” architectures in a single binary. However, it also includes the support for standard x86 architecture. The generic architecture uses a pure C implementation of the APIs and does not use any architecture-specific features.

The specific compiler flags used for building the library with generic configuration are:

```
-O2 -funsafe-math-optimizations -ffp-contract=fast -Wall -Wno-unused-function -Wfatal-errors
```

**Note:** As no architecture specific optimization and vectorized kernels are enabled, performance with the generic architecture may be significantly lower than the architecture-specific implementation.

### 4.4.1.3 Using Dynamic Dispatch

#### Building AOCL-BLAS

Dynamic Dispatch must be enabled while building the AOCL-BLAS library. This is done by building the library for **amdzen** configuration as explained in “Build AOCL-BLAS from Source” on page 24.

#### Code Path Information

Dynamic Dispatch can print debugging information on the selected code path. This is enabled by setting the environment variable **BLIS\_ARCH\_DEBUG=1**.

#### Architecture Selection at Runtime

For most use cases, Dynamic Dispatch will detect the underlying architecture and enable appropriate code paths and optimizations.

However, AOCL-BLAS can be forced to use a specific architecture by setting the environment variable **BLIS\_ARCH\_TYPE** as follows:

```
BLIS_ARCH_TYPE=value <AOCL-BLAS linked application>
```

Where, value = {zen4, zen3, zen2, zen, generic}

You must note the following:

- The code path names are not case sensitive.
- AOCL-BLAS 3.2 and earlier used an enumeration number (across all the supported AOCL-BLAS architecture code paths) to select the code path. The enumeration number could change from release to release as new code paths are added. Thus, it is recommended to avoid setting **BLIS\_ARCH\_TYPE** based on the enumeration number and use a meaningful name as shown above.
- Specifying a particular code path will completely override the automatic selection and thus, the following scenarios are possible:
  - A code path unavailable in the AOCL-BLAS build is being used. This will result in an error message from the AOCL-BLAS library which will then abort.
  - A code path executes instructions unavailable on the processor being used, for example, trying to run the AMD “Zen4” code path (which may use AVX512 instructions) on a AMD “Zen3” or older system. If this happens, the program may stop with an "illegal instruction" error. This may be routine and problem size dependent.

In some circumstances, AOCL-BLAS aborting on an error from **BLIS\_ARCH\_TYPE** being set incorrectly may not be acceptable. If you are building AOCL-BLAS from source, there are two

options to mitigate this issue. One is to change the environment variable used from BLIS\_ARCH\_TYPE to another name, for example:

```
./configure --enable-cblas --prefix=<your-install-dir> --rename-blis-arch-
type=MY_BLIS_ARCH_TYPE amdzen
... make aocl-blas library
... compile program linking with aocl-blas
export BLIS_ARCH_TYPE=zen3
export MY_BLIS_ARCH_TYPE=zen2
./program.exe
```

This will cause *program.exe* (which uses AOCL-BLAS) to ignore the setting of BLIS\_ARCH\_TYPE to zen3. Instead, it will take the value of MY\_BLIS\_ARCH\_TYPE and use the zen2 code path.

Alternatively, the mechanism to allow manual selection of code path can be disabled:

```
./configure --enable-cblas --prefix=<your-install-dir> --disable-blis-arch-type amdzen
```

In this case, Dynamic Dispatch will still occur among the included code paths. However, only by automatic selection based on the code architecture.

## Model Selection at Runtime

Recent AMD “Zen” generations have added more diverse choices of core designs and cache characteristics. For example, Milan and Milan-X variants at AMD “Zen3”; Genoa, Bergamo, and Genoa-X variants at AMD “Zen4”. Some AOCL-BLAS APIs may be tuned differently for these different models. The appropriate model will be selected automatically by Dynamic Dispatch. However, AOCL can be forced to use a specific model by setting the environment variable BLIS\_MODEL\_TYPE as follows:

```
BLIS_MODEL_TYPE=value <AOCL-BLAS linked application>
```

where value = {Milan, Milan-X, Genoa, Bergamo, Genoa-X}

Note the following:

- Different model values correspond to specific BLIS\_ARCH\_TYPE values (either set automatically or explicitly by the user). Thus, Milan and Milan-X correspond to AMD “Zen3”; Genoa, Bergamo, and Genoa-X correspond to AMD “Zen4”.
- Incorrect values of BLIS\_MODEL\_TYPE do not cause an error, the default model type for the selected architecture will be used.
- The number of APIs that have different optimizations by model type is currently very small. Setting this environment variable may provide consistent results across different models if consistency is a higher priority than best performance.

As with BLIS\_ARCH\_TYPE, when building BLAS from source, the name of the environment variable used to set the model type can be changed, for example:

```
./configure --enable-cblas --prefix=<your-install-dir> --rename-blis-model-
type=MY_BLIS_MODEL_TYPE amdzen
```

Disabling the mechanism to allow the manual section of BLAS architecture will also disable the mechanism to allow the manual section of the model.

```
./configure --enable-cblas --prefix=<your-install-dir> --disable-blis-arch-type amdzen
```

### Dynamic Dispatch on non-AMD Architectures

Previous AOCL-BLAS releases identified the processor based on Family, Model, and other cpuid features, and selected the appropriate code path based on the preprogrammed choices. With Dynamic Dispatch, an unknown processor would fall through to the slow "generic" code path, although users could override this by setting BLIS\_ARCH\_TYPE to a suitable value.

In AOCL-BLAS 4.1, additional cpuid tests based on AVX2 and AVX512 instruction support are used to enable AMD “Zen3” or AMD “Zen4” code paths to be selected by default on suitable processors (current or future AMD/Intel processors). The AMD “Zen3” or AMD “Zen4” code paths are not (re-) optimized specifically for these different architectures, but should perform better than the slow "generic" code path.

To be more specific:

- AVX2 support requires AVX2 and FMA3.
- AVX512 support requires AVX512 F, DQ, CD, BW, and VL.

## 4.4.2 AOCL-BLAS - Running the Test Suite

The AOCL-BLAS source directory contains a test suite to verify the functionality of AOCL-BLAS and BLAS APIs. The test suite invokes the APIs with different inputs and verifies that the results are within the expected tolerance limits.

For more information, refer <https://github.com/amd/blis/blob/master/docs/Testsuite.md>.

### 4.4.2.1 Multi-thread Test Suite Performance

Starting from AOCL-BLAS 3.1, the dynamic selection of number of threads is supported. If the number of threads are not specified, AOCL-BLAS uses the maximum number of threads equal to the number of cores available on the system. A higher number of threads result in better performance for medium to large size matrices found in practical use cases.

However, the higher number of threads results in poor performance for very small sizes used by the test and check features. Hence, you must specify the number of threads while running the test/test suite.

The recommended number of threads to run the test suite is 1 or 2.

#### Running Test Suite

Execute the following command to invoke the test suite:

```
$ OMP_NUM_THREADS=2 make test
```

The sample output from the execution of the command is as follows:

```
$:~/blis$ OMP_NUM_THREADS=2 make test
Compiling obj/zen3/testsuite/test_addm.o
Compiling obj/zen3/testsuite/test_addv.o
.
<<< More compilation output >>>
.
Compiling obj/zen3/testsuite/test_xpbym.o
Compiling obj/zen3/testsuite/test_xpbyv.o
Linking test_libblis-mt.x against 'lib/zen3/libblis-mt.a -lm -lpthread -fopenmp -lrt'
Running test_libblis-mt.x with output redirected to 'output.testsuite'
check-blistest.sh: All BLIS tests passed!
Compiling obj/zen3/blastest/cblat1.o
Compiling obj/zen3/blastest/abs.o
.
<<< More compilation output >>>
.
Compiling obj/zen3/blastest/wsfe.o
Compiling obj/zen3/blastest/wsle.o
Archiving obj/zen3/blastest/libf2c.a
Linking cblat1.x against 'libf2c.a lib/zen3/libblis-mt.a -lm -lpthread -fopenmp -lrt'
Running cblat1.x > 'out.cblat1'
.
<<< More compilation output >>>
.
Linking zblat3.x against 'libf2c.a lib/zen3/libblis-mt.a -lm -lpthread -fopenmp -lrt'
Running zblat3.x < './blastest/input/zblat3.in' (output to 'out.zblat3')
check-blastest.sh: All BLAS tests passed!
```

### 4.4.3 Testing/Benchmarking

The AOCL-BLAS source has an API specific test driver and this section explains how to use it for a specific set of matrix sizes.

The source file for this driver is *test/test\_gemm.c* and the executable is *test/test\_gemm\_blis.x*.

Complete the following steps to execute the GEMM tests on specific inputs:

#### Enabling File Inputs

By default, file input/output is disabled (instead it uses start, end, and step sizes). To enable the file inputs, complete the following steps:

1. Open the file *test/test\_gemm.c*.
2. Uncomment the following two macros at the start of the file:
  - a. `#define FILE_IN_OUT`
  - b. `#define MATRIX_INITIALISATION`

#### Building Test Driver

Execute the following commands to build the test driver:

```
$ cd tests
$ make blis
```



## Creating an Input File

The input file accepts matrix sizes and strides in the following format. Each dimension is separated by a space and each entry is separated by a new line.

For example, `m k n cs_a cs_b cs_c`. Where:

- Matrix A is of size `m x k`
- Matrix B is of size `k x n`
- Matrix C is of size `m x n`

This test application (`test_gemm.c`) assumes column-major storage of matrices.

The valid values of `CS_A`, `CS_B`, and `CS_C` for a GEMM operation  $C = \text{beta} * C + \text{alpha} * A * B$ , are as follows:

- `CS_A`  $\geq m$
- `CS_B`  $\geq k$
- `CS_C`  $\geq m$

## Running the Tests

Execute the following commands to run the tests:

```
$ cd tests
$ ./test_gemm_blis.x <input file name> <output file name>
```

An execution sample (with the test driver) for GEMM is as follows:

```
$ cat inputs.txt
200 100 100 200 200 200
10 4 1 100 100 100
4000 4000 400 4000 4000 4000
$ ./test_gemm_blis.x inputs.txt outputs.txt
~~~~~_BLAS m k n cs_a cs_b cs_c gflops
data_gemm_blis 200 100 100 200 200 200 27.211
data_gemm_blis 10 4 1 100 100 100 0.027
data_gemm_blis 4000 4000 400 4000 4000 4000 45.279
$ cat outputs.txt
m k n cs_a cs_b cs_c gflops
200 100 100 200 200 200 27.211
10 4 1 100 100 100 0.027
4000 4000 400 4000 4000 4000 45.279
```

## 4.4.4 AOCL-BLAS Utility APIs

This section explains some of the AOCL-BLAS APIs used to get the AOCL-BLAS library configuration information and for configuring optimization tuning parameters.

**Table 7. AOCL-BLAS Utility APIs**

API	Usages	
bli_info_get_version_str	Returns the version string in the form of “AOCL-BLIS 4.1.0 Build yyyyddmm”.	
bli_info_get_enable_openmp bli_info_get_enable_pthreads bli_info_get_enable_threading	Returns true if OpenMP/pthreads are enabled and false otherwise.	
bli_thread_get_num_threads <sup>1</sup>	Returns the default number of threads used for the subsequent BLAS calls.	
bli_thread_set_num_threads( dim_t n_threads ) <sup>1</sup>	Sets the number of threads for the subsequent BLAS calls.	
bli_thread_set_ways( dim_t jc, dim_t pc, dim_t ic, dim_t jr, dim_t ir ) <sup>1</sup>	Sets the number of threads for different levels of parallelization as per GotoBLAS five loops architecture.	
<b>Notes:</b> 1. Refer <a href="https://github.com/amd/blis/blob/master/docs/Multithreading.md#specifying-multithreading">https://github.com/amd/blis/blob/master/docs/Multithreading.md#specifying-multithreading</a>		

## 4.5 Debugging and Troubleshooting

### 4.5.1 Debugging Build Using GDB

The AOCL-BLAS library can be debugged on Linux using GDB. To enable the debugging support, build the library with the `--enable-debug` flag. Use following commands to configure and build the debug version of AOCL-BLAS:

```
$ cd blis_src
$ ./configure --enable-cblas --enable-debug auto
$ make -j
```

Use the following commands to link the application with the binary and build application with debug support:

```
$ cd blis_src
$ gcc -g -O0 -lpthread -lm -I<path-to-AOCL-BLAS-header> <path-to-AOCL-BLAS-library>/libblis.a test_gemm.c -o test_gemm_blis.x
```

You can debug the application using gdb. A sample output of the gdb session is as follows:

```
$ gdb ./test_gemm_blis.x
GNU gdb (GDB) Red Hat Enterprise Linux 8.2-12.el8
..
..
..
Reading symbols from ./test_gemm_blis.x...done.
(gdb) break bli_gemm_small
Breakpoint 1 at 0x677543: file kernels/zen/3/bli_gemm_small.c, line 110.
(gdb) run
Starting program: /home/dipal/work/blis_dtl/test/test_gemm_blis.x
Using host libthread_db library "/lib64/libthread_db.so.1".
BLIS Library version is : AOCL BLIS 3.1

Breakpoint 1, bli_gemm_small (alpha=0x7fffffffcf40, a=0x2471b30, b=0x7fffffffdd1c0,
beta=0x2465400 <BLIS_ZERO>,
    c=0x4fe66e <bli_obj_equals+300>, cntx=0x7fffffffbb320, cntl=0x0) at kernels/zen/3/
bli_gemm_small.c:110
110     {
(gdb) bt
#0  bli_gemm_small (alpha=0x7fffffffcf40, a=0x2471b30, b=0x7fffffffdd1c0, beta=0x2465400
<BLIS_ZERO>,
    c=0x4fe66e <bli_obj_equals+300>, cntx=0x7fffffffbb320, cntl=0x0) at kernels/zen/3/
bli_gemm_small.c:110
#1  0x000000000007caab6 in bli_gemm_front (alpha=0x7fffffffdd1c0, a=0x7fffffffdd120,
b=0x7fffffffdd080,
    beta=0x7fffffffcf40, c=0x7fffffffcf40, cntx=0x2471b30, rntm=0x7fffffffce50, cntl=0x0)
    at frame/3/gemm/bli_gemm_front.c:83
#2  0x000000000005baf42 in bli_gemmnat (alpha=0x7fffffffdd1c0, a=0x7fffffffdd120,
b=0x7fffffffdd080,
    beta=0x7fffffffcf40, c=0x7fffffffcf40, cntx=0x2471b30, rntm=0x7fffffffce50)
    at frame/ind/oapi/bli_l3_nat_oapi.c:83
#3  0x000000000005474a2 in dgemm_ (transa=0x7fffffffdd363 "N\320a", transb=0x7fffffffdd362
"NN\320a",
    m=0x7fffffffdd36c, n=0x7fffffffdd364, k=0x7fffffffdd368, alpha=0x24733c0, a=0x7ffff53e2040,
lda=0x7fffffffdd378,
    b=0x7ffff355d040, ldb=0x7fffffffdd374, beta=0x2473340, c=0x7ffff16d8040, ldc=0x7fffffffdd370)
    at frame/compat/bla_gemm.c:559
#4  0x00000000000413a1c in main (argc=1, argv=0x7fffffffdd988) at test_gemm.c:321
(gdb)
```

## 4.5.2 Viewing Logs

The AOCL-BLAS library provides Debug and Trace features:

- **Trace Log** identifies the code path taken in terms of the function call chain. It prints the information on the functions invoked and their order.
- **Debug Log** prints the other debugging information, such as values of input parameters, content, and data structures.

The key features of this functionality are as follows:

- Can be enabled/disabled at compile time.

- When these features are disabled at compile time, they do not require any runtime resources and that does not affect the performance.
- Compile time option is available to control the depth of trace/log levels.
- All the traces are thread safe.
- Performance data, such as execution time and gflops achieved, are also printed for xGEMM APIs.

#### 4.5.2.1 Function Call Tracing

The function call tracing is implemented using hard instrumentation of the AOCL-BLAS code. Here, the functions are grouped as per their position in the call stack. You can configure the level up to which the traces must be generated.

Complete the following steps to enable and view the traces:

1. Enable the trace support as follows:

- a. Modify the source code to enable tracing.

```
Open file <aocl-blas folder>/aocl_dtl/aocldtlcf.h
```

- b. Change the following macro from 0 to 1:

```
#define AOCL_DTL_TRACE_ENABLE 0
```

2. Configure the trace depth level.

- a. Modify the source code to specify the trace depth level.

```
Open file <aocl-blas folder>/aocl_dtl/aocldtlcf.h
```

- b. Change the following macro as required. Beginning with Level 5 should be a good compromise in terms of details and resource requirement. The higher the level, the deeper is the call stack. A lower level reduces the depth of the call stack used for a trace generation.

```
#define AOCL_DTL_TRACE_LEVEL AOCL_DTL_LEVEL_TRACE_5
```

3. Build the library as explained in “Build AOCL-BLAS from Source” on page 24.

#### 4. Run the application to generate the trace data.

The trace output file for each thread is generated in the current folder.

The following figure shows a sample running the call tracing function using the test\_gemm application:

```
~/projects/blis_work/blis_gemm_trace/test [amd-staging-rome-2.2 ↑·1|+ 2]
17:51 $ rm *.txt *.rawfile
rm: cannot remove '*.txt': No such file or directory
rm: cannot remove '*.rawfile': No such file or directory
~/projects/blis_work/blis_gemm_trace/test [amd-staging-rome-2.2 ↑·1|+ 2]
17:51 $ export BLIS_NUM_THREADS=4
~/projects/blis_work/blis_gemm_trace/test [amd-staging-rome-2.2 ↑·1|+ 2]
17:51 $ ./test_gemm_blis.x
data_gemm_blis( 1, 1:4 ) = [ 1000 1000 1000 69.27 ];
data_gemm_blis( 2, 1:4 ) = [ 2000 2000 2000 93.31 ];
~/projects/blis_work/blis_gemm_trace/test [amd-staging-rome-2.2 ↑·1|+ 2...4]
17:51 $ ls -l *.txt
-rw-rw-r-- 1 dipal dipal 6428 Jun 10 17:51 P21175_T21175_aocldtl_trace.txt
-rw-rw-r-- 1 dipal dipal 6142 Jun 10 17:51 P21175_T21176_aocldtl_trace.txt
-rw-rw-r-- 1 dipal dipal 6142 Jun 10 17:51 P21175_T21177_aocldtl_trace.txt
-rw-rw-r-- 1 dipal dipal 6142 Jun 10 17:51 P21175_T21178_aocldtl_trace.txt
~/projects/blis_work/blis_gemm_trace/test [amd-staging-rome-2.2 ↑·1|+ 2...4]
17:51 $
```

**Figure 1. Sample Run of Function Call Tracing**

The trace data for each thread is saved in the file with appropriate naming conventions. The *.txt* extension is used to signify the readable file:

*P<process id>\_T<thread id>\_aocldtl\_trace.txt*

#### 5. View the trace data.

The output of the call trace is in a readable format, you can open the file in any of the text editors. The first column shows the level in call stack for the given function.

#### 4.5.2.2 Debug Logging

The debug logging works very similar to the function call tracing and uses the same infrastructure. However, it can be enabled independent of the trace feature to avoid cluttering of the overall debugging information. This feature is primarily used to print the input values of the AOCL-BLAS APIs. Additionally, it can also be used to print any arbitrary debugging data (buffers, matrices, arrays, or text).

Complete the following steps to enable and view the debug logs:

##### 1. Enable the debug log support as follows:

- a. Modify the source code to enable debug logging.

Open file <aocl-blas folder>/aocl\_dtl/aocldtlcf.h

- b. Change the following macro from 0 to 1:

```
#define AOCL_DTL_LOG_ENABLE 0
```

2. Configure the trace depth level.

- a. Modify the source code to specify the debug log depth level.

```
Open file <aocl-blas folder>/aocl_dtl/aocldtlcf.h
```

- b. Change the following macro as required. Beginning with Level 5 should be a good compromise in terms of details and resource requirement. The higher the level (maximum is 10), the deeper is the call stack. A lower level reduces the depth of the call stack used for a trace generation.

```
#define AOCL_DTL_TRACE_LEVEL AOCL_DTL_LEVEL_TRACE_5
```

3. Build the library as explained in “Build AOCL-BLAS from Source” on page 24.

4. Run the application to generate the trace data.

The trace output files for each thread is generated in the current folder.

The following figure shows a sample running of AOCL-BLAS with the debug logs enabled using the test\_gemm application:

```
~/projects/blis_work/blis_gemm_trace/test [amd-staging-milan-3.0|+ 3...1]
09:52 $ rm *.txt
~/projects/blis_work/blis_gemm_trace/test [amd-staging-milan-3.0|+ 3]
09:52 $ ./test_gemm_blis.x
BLIS Library version is : AOCL-3.0
data_gemm_aocl( 1, 1:4 ) = [ 1000 1000 1000 98.03 ];
data_gemm_aocl( 2, 1:4 ) = [ 2000 2000 2000 100.55 ];
~/projects/blis_work/blis_gemm_trace/test [amd-staging-milan-3.0|+ 3...1]
09:52 $ ls -al *.txt
-rw-rw-r-- 1 dipal dipal 582 Nov 9 09:52 P18597_T0_aocldtl_log.txt
~/projects/blis_work/blis_gemm_trace/test [amd-staging-milan-3.0|+ 3...1]
09:52 $
```

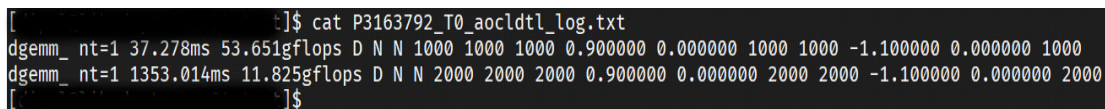
**Figure 2. Sample Run with Debug Logs Enabled**

The debug logs for each thread are saved in the file with appropriate naming conventions. The .txt extension is used to signify the readable file:

*P<process id>\_T<thread id>\_aocldtl\_log.txt*

## 5. View the debug logs.

The output of the debug logs is in a readable format, you can open the file in any of the text editors. The following figure shows the sample output for one of the threads of test\_gemm application:



```

]$ cat P3163792_T0_aocldtl_log.txt
dgemm_ nt=1 37.278ms 53.651gflops D N N 1000 1000 1000 0.900000 0.000000 1000 1000 -1.100000 0.000000 1000
dgemm_ nt=1 1353.014ms 11.825gflops D N N 2000 2000 2000 0.900000 0.000000 2000 2000 -1.100000 0.000000 2000
]$

```

**Figure 3. Debug Logs Showing Input Values of GEMM**

### 4.5.2.3 Usages and Limitations

The debug and trace logs have the following usages and limitations:

- When tracing is enabled, there could be a significant drop in the performance.
- Only a function that has the trace feature in the code can be traced. To get the trace information for any other function, the source code must be updated to add the trace/log macros in them.
- The call trace and debug logging is a resource-dependent process and can generate a large size of data. Based on the hardware configuration (the disk space, number of cores and threads) required for the execution, logging may result in a sluggish or non-responsive system.

### 4.5.3 Checking AOCL-BLAS Operation Progress

The AOCL libraries may be used to perform lengthy computations (for example, matrix multiplications and solver involving large matrices). These operations/computations may go on for hours.

AOCL Progress feature provides mechanism for the application to check the computation progress. The AOCL libraries (AOCL-BLAS and AOCL-LAPACK) periodically updates the application with progress made through a callback function.

#### Usage

The application must define the callback function in a specific format and register it with the AOCL library.

#### Callback Definition

The callback function prototype must be as defined as given follows:

```

int AOCL_BLIS_progress(
const char* const api,
const int lapi,
const dim_t progress,
const dim_t current_thread,
const dim_t total_threads
)

```

However, you can modify the function name as per your preference.

The following table explains different parameters passed to the callback function:

**Table 8. Callback Parameters**

Parameter	Purpose
api	Name of the API running currently
lapi	Length of the API name string (*api)
progress	Linear progress made in current thread presently
current_thread	Current thread ID
total_threads	Total number of threads used to performance the operation

### Callback Registration

The callback function must be registered with the library for reporting the progress. Each library has its own callback registration function. The registration can be done by calling:

**AOCL\_BLIS\_set\_progress(AOCL\_progress);** // for AOCL-BLAS

### Example

The library only invokes the callback function at appropriate intervals, it is up to the user to consume this information appropriately. The following example shows how to use it for printing the progress to a standard output:

```
int AOCL_BLIS_progress(
const char* const api,
const int lapi,
const dim_t progress,
const dim_t current_thread,
const dim_t total_threads
)
{
    printf("\n%s, total thread = %lld, processed %lld element by thread %lld.",
        api, total_threads, progress, current_thread);
    return 0;
}
```

Register the callback with:

**AOCL\_BLIS\_set\_progress(AOCL\_progress);** // for AOCL-BLAS

The result is displayed in following format (output truncated):

```
BLIS_NUM_THREADS=5 ./test_gemm_blis.x
dgemm, total thread = 5, processed 11796480 element by thread 4.
dgemm, total thread = 5, processed 17694720 element by thread 0.
dgemm, total thread = 5, processed 5898240 element by thread 2.
dgemm, total thread = 5, processed 20643840 element by thread 0.
dgemm, total thread = 5, processed 14745600 element by thread 3.
dgemm, total thread = 5, processed 14745600 element by thread 4.
```



## Limitations

- The feature only shows if the operation is progressing or not, it doesn't provide an estimate/percentage compilation status.
- A separate callback must be registered for AOCL-BLAS, AOCL-LAPACK, and AOCL-ScaLAPACK.

## 4.6 Build AOCL-BLAS from Source on Windows

GitHub URL: <https://github.com/amd/blis>

AOCL-BLAS uses CMake along with Microsoft Visual Studio for building binaries from the sources on Windows. The following sections explain the GUI and command-line schemes of building the binaries and test suite.

### Prerequisites

- Windows 10/11 or Windows Server 2019/2022
- LLVM 13/14 for AMD “Zen3” and AMD “Zen4” support (or LLVM 11 for AMD “Zen2” support)
- LLVM plug-in for Microsoft Visual Studio (if latest version of LLVM is installed separately, this plugin enables linking Visual Studio with the installed LLVM toolchain)
- CMake 3.0 through 3.23.3
- Microsoft Visual Studio 2019 (build 16.8.7) through 2022 (build 17.3.2)
- Microsoft Visual Studio tools (as shown in Figure 4):
  - Python development
  - Desktop development with C++: C++ Clang-Cl for v142 build tool (x64/x86)

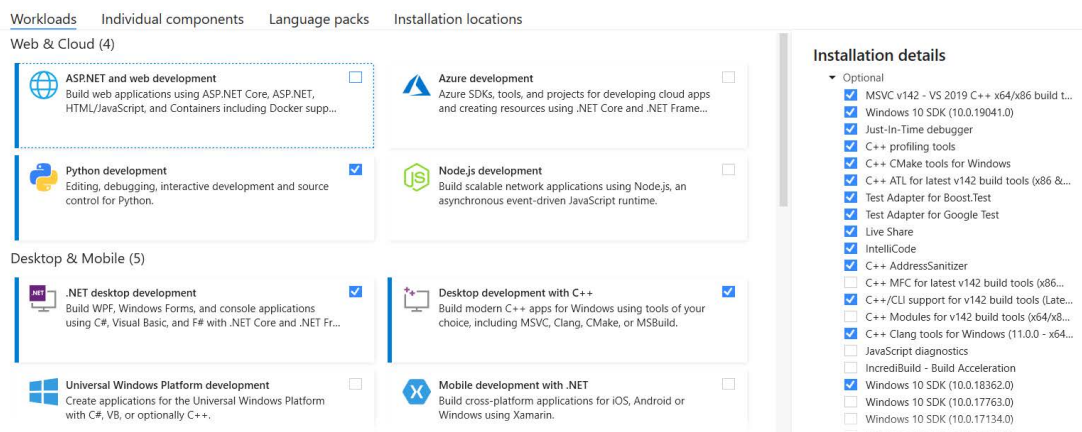


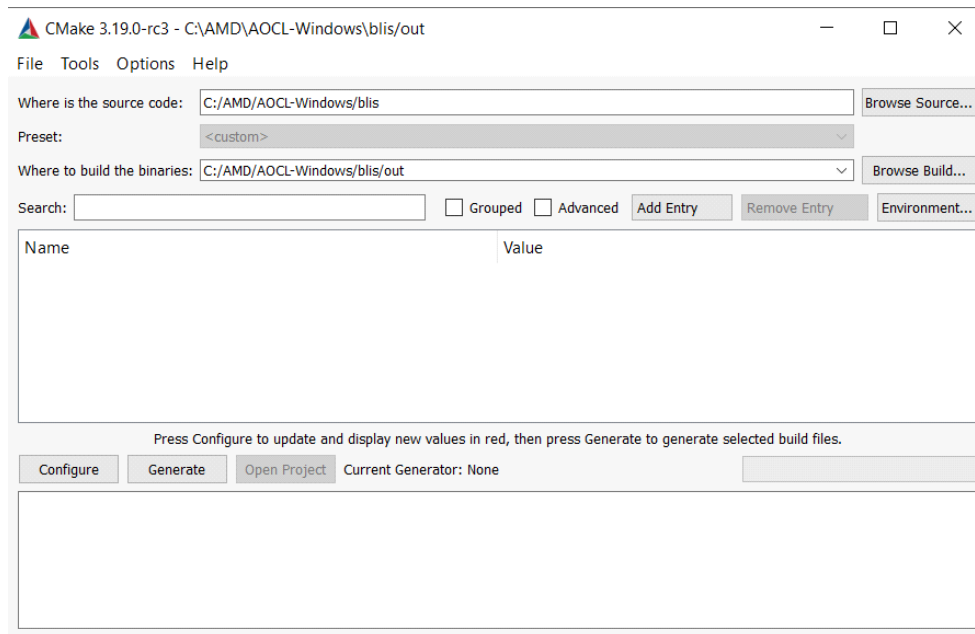
Figure 4. Microsoft Visual Studio Prerequisites

## 4.6.1 Building AOCL-BLAS using GUI

### 4.6.1.1 Preparing Project with CMake GUI

Complete the following steps in the CMake GUI:

1. Set the **source** (folder containing AOCL-BLAS source code) and **build** (folder in which the project files will be generated, for example, **out**) folder paths as shown in the following figure:

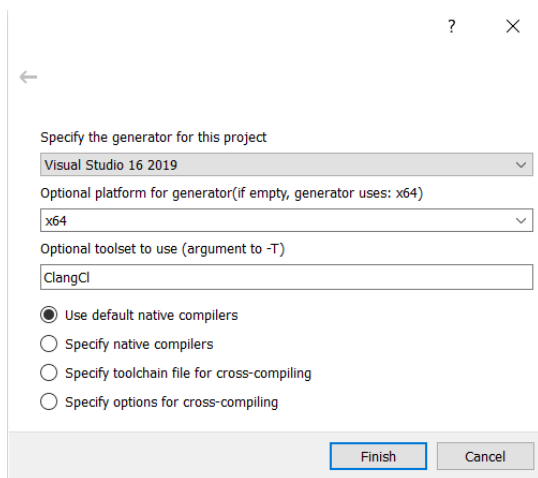


**Figure 5. CMake Source and Build Folders**

It is not recommended to use the folder named **build** since **build** is used by Linux build system.

2. Click on the **Configure** button to prepare the project options.

3. Set the generator to **Visual Studio 16 2019** or **Visual Studio 17 2022** and the compiler to **ClangCl** or **LLVM** as shown in the following figure:



**Figure 6. Set Generator and Compiler**

4. Update the options based on the project requirements. All the available options are listed in the following table:

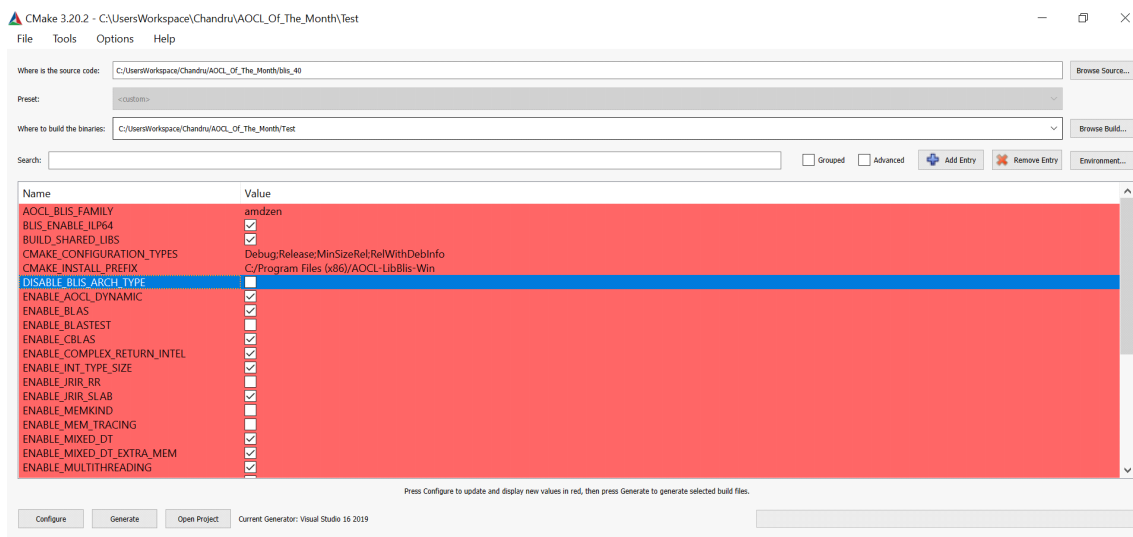
**Table 9. CMake Config Options**

Feature	CMake Parameter
AMD CPU architecture	AOCL_BLIS_FAMILY:STRING=zen/zen2/zen3
Enable verbose mode	ENABLE_VERBOSE=ON
Shared library	BUILD_SHARED_LIBS=ON
Static library	BUILD_SHARED_LIBS=OFF ENABLE_AOCL_DYNAMIC=OFF
Debug/Release build type	CMAKE_BUILD_TYPE=Debug/Release
Dynamic Dispatcher	AOCL_BLIS_FAMILY:STRING=amdzen
Enable single threading	ENABLE_MULTITHREADING=OFF ENABLE_AOCL_DYNAMIC=OFF
Enable multi-threading with OpenMP and AOCL dynamic enabled	ENABLE_MULTITHREADING=ON ENABLE_OPENMP=ON ENABLE_AOCL_DYNAMIC=ON
Enable multi-threading with OpenMP and AOCL dynamic disabled	ENABLE_MULTITHREADING=ON ENABLE_OPENMP=ON ENABLE_AOCL_DYNAMIC=OFF
Enable BLAS/CBLAS support	ENABLE_BLAS=ON ENABLE_CBLAS=ON

**Table 9. CMake Config Options**

Feature	CMake Parameter
Enable 32-bit integer size in BLIS and BLAS APIs	BLIS_ENABLE_ILP64=OFF ENABLE_INT_TYPE_SIZE=OFF
Enable 64-bit integer size in BLIS and BLAS APIs	BLIS_ENABLE_ILP64=ON ENABLE_INT_TYPE_SIZE=ON
Flags that are enabled by default	ENABLE_JRIR_SLAB ENABLE_PBA_POOLS ENABLE_SBA_POOLS ENABLE_MIXED_DT ENABLE_MIXED_DT_EXTRA_MEM ENABLE_SUP_HANDLING ENABLE_PRAGMA_OMP_SIMD
Flags that are disabled by default	ENABLE_JRIR_RR ENABLE_MEM_TRACING ENABLE_MEMKIND ENABLE_SANDBOX
Use APIs without trailing underscore	ENABLE_NO_UNDERSCORE_API
Enable uppercase APIs	ENABLE_UPPERCASE_API
Absolute path to the OpenMP library, including the library name	OpenMP_libomp_LIBRARY
Disable forced code path selection using the environment variable BLIS_ARCH_TYPE	DISABLE_BLIS_ARCH_TYPE

5. To generate the Microsoft Visual Studio project in the **out** folder, click on the **Generate** button as shown in the following figure:



**Figure 7. CMake Configure and Generate Project Settings**

#### 4.6.1.2 Building the Project in Visual Studio GUI

Complete the following steps in the Microsoft Visual Studio GUI:

1. Open the project generated by CMake (build folder) in “Preparing Project with CMake GUI” on page 50.
2. To generate AOCL-BLAS binaries, build the **AOCL-LibBlis-Win** project.

The library files will generate in the **bin** folder based on the project settings.

For example, *blis/bin/Release/AOCL-LibBlis-Win.dll* or *AOCL-LibBlis-Win.lib*

### 4.6.2 Building AOCL-BLAS using Command-line Arguments

The project configuration and build procedures can be triggered from the command prompt as well. The corresponding steps are described in the following sections.

#### 4.6.2.1 Configuring the Project in Command Prompt

In the AOCL-BLAS project folder, create a folder **out**. Open the command prompt in this directory and run the following command to configure the project:

```
cmake -S .. -B . -G "Visual Studio 16 2019" -DCMAKE_BUILD_TYPE=Release
-DAOCL_BLIS_FAMILY:STRING=amdzen -DBUILD_SHARED_LIBS=ON -DENABLE_MULTITHREADING=ON
-DENABLE_OPENMP=ON -DENABLE_COMPLEX_RETURN_INTEL=ON -DOpenMP_libomp_LIBRARY="C:\Program
Files\LLVM\lib\libomp.lib"
-DENABLE_AOCL_DYNAMIC=ON -TClangCL
```

You can refer Table 9 and update the parameter options in the command according to the project requirements.

#### 4.6.2.2 Building the Project in Command Prompt

Open command prompt in the *blis\out* directory. Invoke CMake with the build command with release or debug option. For example:

```
cmake --build . --config Release
```

The library files would be generated in the **Release** or **Debug** folder based on the project settings.

### 4.6.3 Building and Running the Test Suite

The Microsoft Visual Studio projects for individual tests and the test suite are generated as a part the CMake generate step. You can build the test projects from Microsoft Visual Studio GUI or command prompt as described in the previous sections.

If you prefer to build the application or the test suite executable with the pre-built static library (from the package) on Windows, both the instances of "#define BLIS\_ENABLE\_SHARED" must be commented out in the header file *blis.h*.

#### 4.6.3.1 Running Individual Tests

Copy the relevant input files for the tests from *blis\bench* to the *blis\bin\release* folder. Run the tests from the command prompt as follows:

```
Release> TestGemm.exe inputgemm.txt output.txt
```

#### 4.6.3.2 Running the Test Suite

Copy the input files *input.global.general* and *input.global.operations* for the tests from *blis\test* to the release folder. The tests can be run from command prompt as follows:

```
Release> test_libblis.exe
```

#### 4.6.3.3 Running Multi-thread Tests

Complete the following steps to run the multi-thread tests:

1. Copy the relevant input files for the tests from *blis\testsuite* or *blis\bench* to the *blis\bin\release* folder.
2. Copy *libomp.lib* and *libomp.dll* respectively from the Microsoft Visual Studio folders *\VC\Tools\Llvm\lib* and *\VC\Tools\Llvm\bin* to the *blis\bin\release* folder.
3. Set the threading environment variables in the same command prompt session as the test runs.

For example:

```
Release> set BLIS_NUM_THREADS=x (x could be no of threads)
Release> set OMP_PROC_BIND=spread
Release> TestGemm.exe inputgemm.txt output.txt
```

## 4.7 LPGEMM in AOCL-BLAS

### 4.7.1 Add-on in AOCL-BLAS

An add-on in AOCL-BLAS provides additional APIs, operations, and/or implementations that may be useful to certain users. It can be a standalone extension of AOCL-BLAS that does not depend on any other add-on, although add-ons may utilize existing functionality or kernels within the core framework.

An add-on should never provide APIs that conflict with the interfaces belonging to the BLIS typed or object API. Thus, a properly constructed/functioning add-on would never interfere with or change the core BLIS functionality or the standard BLAS and CBLAS APIs.

Low Precision GEMM (LPGEMM) APIs are added as an add-on feature with the name `aocl_gemm` in AOCL-BLAS 4.1 which are used in Inference of Deep Neural Networks (DNN) applications. For example, Low Precision DNN uses the input as image pixels that are unsigned 8-bit (u8) and quantized pre-trained weights of signed 8-bits (s8) width. They produce signed 32-bit or downsampled/quantized 8-bit output.

At the same time, these APIs are expected to utilize the architecture features such as AVX512VNNI instructions designed to take the inputs in u8, s8; produce an output in s32 and produce high throughput. Similarly, AVX512BF16 based instructions expects input in Brain Floating Point (bfloat16) type to provide higher throughput with less precision than 32-bit.

### 4.7.2 API Naming and Arguments

LPGEMM APIs starts with the prefix "`aocl_gemm_`" and follows the data type of input matrix A, B, Accumulation type, and output matrix C.

For example, `aocl_gemm_u8s8s32os32()` API expects input matrix is unsigned 8bit (u8) and signed 8 bit (s8), accumulation type is signed 32-bit (s32) and output matrix type is signed 32-bit (o s32).

### 4.7.3 Post-operations

The low precision GEMM operations are highly useful in AI applications, where the precision requirements can be traded with performance. In DNN applications element-wise operations, such as adding bias, clip the output, ReLU, and GeLU are performed on the GEMM output which are referred here as post-operations (post-ops).

In LPGEMM, these post-ops are fused with the GEMM operation to avoid loading of data again into registers and thereby, improving the performance. In the LPGEMM APIs, an additional argument is added for the user to provide information about the post-ops needed to perform after the GEMM operation.

## 4.7.4 Supported APIs in aocl\_gemm

### 4.7.4.1 GEMM APIs and Supported Post-ops

**Table 10. GEMM APIs and Supported Post-ops**

Arch Features Required	API/Pos-ops	Add bias	ReLU	PReLU	GeLU-Tanh	GeLU-Erf	Down Scale	CLIP
AVX512VNNI	aocl_gemm_u8s8s32os32	Yes	Yes	Yes	Yes	Yes	No	Yes
	aocl_gemm_u8s8s32os8	Yes	Yes	Yes	Yes	Yes	Yes	Yes
	aocl_gemm_s8s8s32os32	Yes	Yes	Yes	Yes	Yes	No	Yes
	aocl_gemm_s8s8s32os8	Yes	Yes	Yes	Yes	Yes	Yes	Yes
AVX2	aocl_gemm_u8s8s16os16	Yes	Yes	Yes	Yes	Yes	No	Yes
	aocl_gemm_u8s8s16os8	Yes	Yes	Yes	Yes	Yes	Yes	Yes
	aocl_gemm_s8s8s16os16	Yes	Yes	Yes	Yes	Yes	No	Yes
	aocl_gemm_s8s8s16os8	Yes	Yes	Yes	Yes	Yes	Yes	Yes
AVX512BF16	aocl_gemm_bf16bf16f32of32	Yes	Yes	Yes	Yes	Yes	No	Yes
	aocl_gemm_bf16bf16f32obf16	Yes	Yes	Yes	Yes	Yes	Yes	Yes
AVX512	aocl_gemm_f32f32f32of32	Yes	Yes	Yes	Yes	Yes	No	Yes

### 4.7.4.2 Utility APIs in aocl\_gemm Add-on

LPGEMM APIs supports reordering the entire input matrix before calling GEMM and on the go packing, where GEMM API takes care of packing of matrix internally. The following utility APIs are used to reorder input weight matrix before calling GEMM:

**Table 11. Utility APIs in aocl\_gemm Add-on**

API	Description
aocl_get_reorder_buff_size_XXX XXXXX()	Returns buffer size required to reorder an input matrix, where XXXXXXXX corresponds to each of the data type combinations specified in <a href="#">Table 10</a> . For example, u8s8s32os32.
aocl_reorder_XXXXXXXXX ()	Reorders the given input and writes into output buffer.
aocl_gelu_tanh_f32()	Performs tanh operation on each element of the given input buffer and writes in the output buffer.
aocl_gelu_erf_f32()	Performs tanh operation on each element of the given input buffer and writes in the output buffer.
aocl_softmax_f32()	Performs tanh operation on each element of the given input buffer and writes in the output buffer.



### 4.7.5 Enabling aocl\_gemm Add-on

Enabling aocl\_gemm add-on while building AOCL-BLAS from Source:

- Building with GCC:

```
./configure -a aocl_gemm --enable-cblas --enable-threading=Openmp
--prefix=<your-install-dir> CC=gcc CXX=g++ [auto | amdzen]
```

- Building with AOCC:

```
./configure -a aocl_gemm --enable-cblas --enable-threading=openmp
--prefix=<your-install-dir> CC=clang CXX=clang++ [auto | amdzen]
```

- The aocl\_gemm add-on feature is not supported on Windows.
- Refer to *blis.h* file for all the prototypes of LPGEMM APIs.
- Some LPGEM APIs are supported only when the architecture features, such as avx512vnni and avx512bf16 are available in the machine as mentioned in [Table 10](#). The APIs returns without doing anything when those features are not available.

### 4.7.6 Sample Application 1

The following sample application is to use the LPGEMM APIs without post-ops:

```
//$gcc test_LPGEMM.c -o test_lpgemm -I/aocl-blis_install_directory/include/blis
// -L/aocl-blis_install_directory/lib/ -lblis-mt -lm

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "BLIS.h"

// Example program to demonstrate LPGEMM API usage.
// aocl_gemm_u8s8s32os32 (A:uint8_t, B:int8_t, C:int32_t) used here.
int main()
{
    dim_t m = 1024;
    dim_t n = 1024;
    dim_t k = 1024;

    // Leading dimensions for row major matrices.
    dim_t lda = k;
    dim_t ldb = n;
    dim_t ldc = n;
```

```

uint8_t* a = ( uint8_t* ) bli_malloc_user( sizeof( uint8_t ) * m * k );
int8_t* b = ( int8_t* ) bli_malloc_user( sizeof( int8_t ) * n * k );
int32_t* c = ( int32_t* ) bli_malloc_user( sizeof( int32_t ) * m * n );

// Functions to fill the matrices with data can be added here.

int32_t alpha = 2;
int32_t beta = 9;

char storage = 'r'; // Row major. Use 'c' for column major.
char transa = 'n'; // No transpose. Transpose not supported.
char transb = 'n';
char reordera = 'n';
char reorderb = 'r'; // Reorder B matrix, equal to packing entire B matrix.

aocl_gemm_u8s8s32os32
(
    storage, transa, transb,
    m, n, k,
    alpha,
    a, lda, reordera,
    b, ldb, reorderb,
    beta,
    c, ldc,
    NULL
);
if ( a != NULL )
{
    bli_free_user( a );
}
if ( b != NULL )
{
    bli_free_user( b );
}
if ( c != NULL )
{
    bli_free_user( c );
}

return 0;
}

```

### 4.7.7 Sample Application 2

The following sample application is to use the LPGEMM Downscale APIs with post-ops:

```
// $gcc test_lpgemm.c -o test_LPGEMM -I/aocl-blis_install_directory/include/blis
// -L/aocl-blis_install_directory/lib/ -lBLIS-mt -lm

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "BLIS.h"
// Example program to demonstrate LPGEMM downscale API with post-ops usage.
// aocl_gemm_u8s8s32os8 (A:uint8_t, B:int8_t, C:int8_t) used here.
// 3 post-ops - bias + gelu_tanh + clip used here.
int main()
{
    dim_t m = 1024;
    dim_t n = 1024;
    dim_t k = 1024;

    // Leading dimensions for row major matrices.
    dim_t lda = k;
    dim_t ldb = n;
    dim_t ldc = n;

    uint8_t* a = ( uint8_t* ) bli_malloc_user( sizeof( uint8_t ) * m * k );
    int8_t* b = ( int8_t* ) bli_malloc_user( sizeof( int8_t ) * n * k );
    int8_t* c = ( int8_t* ) bli_malloc_user( sizeof( int8_t ) * m * n );

    // Functions to fill the matrices with data can be added here.
    int32_t alpha = 2;
    int32_t beta = 9;

    char storage = 'r'; // Row major. Use 'c' for column major.
    char transa = 'n'; // No transpose. Transpose not supported.
    char transb = 'n';
    char reordera = 'n';
    char reorderb = 'r'; // Reorder B matrix, equal to packing entire B matrix.

    // Initialize post-ops struct.
    aocl_post_op* post_ops = NULL;
    post_ops = ( aocl_post_op* ) malloc( sizeof( aocl_post_op ) );
    // Downscale parameters need to be passed as a post-op, even
```

```

// if a downscale specific api is invoked.
dim_t max_post_ops_seq_length = 4; // bias+gelu_tanh+clip+downscale
post_ops->seq_vector = ( AOCL_POST_OP_TYPE* )
    malloc
    (
        max_post_ops_seq_length *
        sizeof( AOCL_POST_OP_TYPE )
    );
// Bias
post_ops->seq_vector[0] = BIAS;
// Need to output accumulation (int32_t) type for bias.
post_ops->bias.bias = malloc( n * sizeof( int32_t ) );
// Add function to fill bias array here.
post_ops->seq_vector[1] = ELTWISE; // For gelu_tanh
post_ops->seq_vector[2] = ELTWISE; // For clip
// 2 element wise post-ops, need to allocate dynamically.
post_ops->eltwise = malloc( 2 * sizeof( aocl_post_op_eltwise ) );

// Gelu tanh.
( post_ops->eltwise + 0 )->is_power_of_2 = FALSE;
( post_ops->eltwise + 0 )->scale_factor = NULL;
( post_ops->eltwise + 0 )->algo.alpha = NULL;
( post_ops->eltwise + 0 )->algo.beta = NULL;
( post_ops->eltwise + 0 )->algo.algo_type = GELU_TANH;

// Clip.
( post_ops->eltwise + 1 )->is_power_of_2 = FALSE;
( post_ops->eltwise + 1 )->scale_factor = NULL;
// Min bound is represented by alpha.
( post_ops->eltwise + 1 )->algo.alpha = malloc( sizeof( int32_t ) );
// Max bound is represented by beta.
( post_ops->eltwise + 1 )->algo.beta = malloc( sizeof( int32_t ) );
//Set some min/max bounds.
*( ( int32_t* ) ( post_ops->eltwise + 1 )->algo.alpha ) = (int32_t) (-64);
*( ( int32_t* ) ( post_ops->eltwise + 1 )->algo.beta ) = (int32_t) ( 3 );
( post_ops->eltwise + 1 )->algo.algo_type = CLIP;

// Downscale
post_ops->seq_vector[3] = SCALE;
post_ops->sum.is_power_of_2 = FALSE;
post_ops->sum.scale_factor = NULL;
post_ops->sum.buff = NULL;
post_ops->sum.zero_point = NULL;
post_ops->sum.scale_factor = malloc( n * sizeof( float ) );

```

```
// Add function to fill downscale array here.
post_ops->seq_length = 4;
aocl_gemm_u8s8s32os8
(
    storage, transa, transb,
    m, n, k,
    alpha,
    a, lda, reordera,
    b, ldb, reorderb,
    beta,
    c, ldc,
    post_ops
);

if ( post_ops->sum.scale_factor != NULL )
{ free( post_ops->sum.scale_factor ); }

if ( ( post_ops->eltwise + 1 )->algo.alpha != NULL )
{ free( ( post_ops->eltwise + 1 )->algo.alpha ); }

if ( ( post_ops->eltwise + 1 )->algo.beta != NULL )
{ free( ( post_ops->eltwise + 1 )->algo.beta ); }

if ( post_ops->eltwise != NULL )
{ free( post_ops->eltwise ); }

if ( post_ops->bias.bias != NULL )
{ free( post_ops->bias.bias ); }

if ( post_ops->seq_vector != NULL )
{ free( post_ops->seq_vector ); }

if ( post_ops != NULL )
{ free( post_ops ); }

if ( a != NULL )
{ bli_free_user( a ); }

if ( b != NULL )
{ bli_free_user( b ); }

if ( c != NULL )
{ bli_free_user( c ); }

return 0;
}
```

---

## Chapter 5 AOCL-LAPACK

---

AOCL-LAPACK is a high performant implementation of Linear Algebra PACKage (LAPACK). LAPACK provides routines for solving systems of linear equations, least-squares problems, eigenvalue problems, singular value problems, and the associated matrix factorizations. It is extensible, easy to use, and available under an open-source license. Applications relying on standard Netlib LAPACK interfaces can utilize AOCL-LAPACK with virtually no changes to their source code. AOCL-LAPACK supports C, Fortran, and C++ template interfaces (for a subset of APIs) for the LAPACK APIs.

AOCL-LAPACK is based on libFLAME, which was originally developed by current and former members of the *Science of High-Performance Computing (SHPC)* group in the *Institute for Computational Engineering and Sciences* at *The University of Texas at Austin* under the project name libflame. The upstream libFLAME repository is available on GitHub (<https://github.com/flame/libflame>). AMD is actively optimizing key routines in libFLAME as a part of the AOCL-LAPACK library, for AMD “Zen”-based architectures in the "amd" fork of libFLAME hosted on AMD GitHub.

From AOCL 4.1, AOCL-LAPACK is compatible with LAPACK 3.11.0 specification. In combination with the AOCL-BLAS library, which includes optimizations for the AMD “Zen”-based processors, AOCL-LAPACK enables running high performing LAPACK functionalities on AMD platforms.

### 5.1 Installing on Linux

AOCL-LAPACK can be installed from source or pre-built binaries.

#### 5.1.1 Building AOCL-LAPACK from Source

GitHub URL: <https://github.com/amd/libflame>

**Note:** *Building AOCL-LAPACK does not require linking to AOCL-BLAS or any other BLAS library. The applications which use AOCL-LAPACK must link to AOCL-BLAS (or other BLAS libraries) for the BLAS functionalities.*

##### Prerequisites

The following dependencies must be met for installing AOCL-LAPACK:

- Target CPU ISA supporting AVX2 and FMA
- Python versions 3.4 and 3.6
- GNU Make 4.2
- GCC, g++, and Gfortran (versions 12.2 through 13.1)
- AOCL-Utills library

## Build Steps

From AOCL 4.1, AOCL-LAPACK supports compiling the library using CMake build system in addition to the existing configure script method on Linux. Both the approaches to build the library are explained in this section.

Complete the following steps to build AOCL-LAPACK from source:

1. Clone the Git repository (<https://github.com/amd/libflame.git>).
2. Compile AOCL-LAPACK source.

### Method 1: Using Configure/Makefile

1. Run the configure script. An example below shows the recommended options to be used when compiling on AMD “Zen”-based processors.

- With GCC (default)

Using 32-bit Integer (LP64)

```
$ ./configure --enable-amd-flags --prefix=<your-install-dir>
```

Using 64-bit Integer (ILP64)

```
$ ./configure --enable-amd-flags -enable-ilp64 --prefix=<your-install-dir>
```

- With AOCC

```
$ export CC=clang
$ export FC=flang
$ export FLIBS="-lflang"
```

Using 32-bit Integer (LP64)

```
$ ./configure --enable-amd-aocc-flags --prefix=<your-install-dir>
```

Using 64-bit Integer (ILP64)

```
$ ./configure --enable-amd-aocc-flags -enable-ilp64 --prefix=<your-install-dir>
```

2. Make and install using the following commands:

```
$ make -j
$ make install
```

By default, without the configure option **prefix**, the library will be installed in *\$HOME/flame*.

### Method 2: Using CMake

1. Create a new build directory, for example, newbuild:

```
$ mkdir newbuild
$ cd newbuild
```

## 2. Run the following command to configure the project:

- With GCC (default):

**Using 32-bit Integer (LP64)**

```
cmake ../ -DENABLE_AMD_FLAGS=ON -DCMAKE_INSTALL_PREFIX=<your-install-dir>
```

**Using 64-bit Integer (ILP64)**

```
cmake ../ -DENABLE_ILP64=ON -DENABLE_AMD_FLAGS=ON -DCMAKE_INSTALL_PREFIX=<your-install-dir>
```

- With AOCC:

```
export CC=clang
export CXX=clang++
export FC=flang
export FLIBS="-lflang"
```

**Using 32-bit Integer (LP64)**

```
cmake ../ -DENABLE_AMD_AOCC_FLAGS=ON -DCMAKE_INSTALL_PREFIX=<your-install-dir>
```

**Using 64-bit Integer (ILP64)**

```
cmake ../ -DENABLE_ILP64=ON -DENABLE_AMD_AOCC_FLAGS=ON -DCMAKE_INSTALL_PREFIX=<your-install-dir>
```

Shared library is turned on by default. To generate static library, provide the additional option:

**-DBUILD\_SHARED\_LIBS=OFF**

## 3. Compile the library using the following command:

```
cmake --build . -j
```

or

```
make -j
```

This will generate *libflame.a/libflame.so* library in the *lib* directory

## Linking with AOCL-Utills Library

AOCL-LAPACK requires the AOCL-Utills static library "libaoclutils" for certain functions including CPU architecture detection at runtime. The AOCL-LAPACK build system, by default, automatically links with libaoclutils library. It is done by downloading the source of libaoclutils from AMD GitHub, compiling it and linking/merging with the AOCL-LAPACK library. However, you can provide an external path for libaoclutils binary and header files through separate flags. In this scenario, the build system will use the user provided library and does not download libaoclutils source.

For Configure/Makefile build system to explicitly set the libaoclutils path, set the flags as follows:

```
$ export CFLAGS="-I<path to libaoclutils include directory>"
$ configure <standard recommended flags as mentioned earlier>
$ make LIBAOCLUTILS_LIBRARY_PATH="<path to libaoclutils library>" -j
```



For CMake build system to explicitly set libaoclutils path, set the flags `LIBAOCLUTILS_LIBRARY_PATH` and `LIBAOCLUTILS_INCLUDE_PATH` as follows:

```
$ cmake ../ -DENABLE_AMD_FLAGS=ON -DCMAKE_INSTALL_PREFIX=<your-install-dir> -
-DLIBAOCLUTILS_LIBRARY_PATH=<path to libaoclutils library> -DLIBAOCLUTILS_INCLUDE_PATH=<path to
libaoclutils header files>
```

### Additional Notes on Configuration Options

1. By default, the configuration options `--enable-amd-flags` and `--enable-amd-aocc-flags` enable multi-threading using OpenMP for the selected APIs in AOCL-LAPACK. To disable multi-threading, use the configure option `--enable-multithreading=no`.

Example:

```
$ ./configure --enable-amd-flags --enable-multithreading=no
```

or

```
$ ./configure --enable-amd-aocc-flags --enable-multithreading=no
```

Similarly, for CMake, use the flag `ENABLE_MULTITHREADING` to set multi-threading ON/OFF.

2. To support binary portability across different architectures, the default compiler flags are set to `-mtune=native -mavx2 -mfma -O3`.

This requires AVX2 and Fused Multiply Accumulate (FMA) support from the target CPU as mentioned in the Prerequisites section.

For enabling further optimizations, such as enabling AVX, AVX2, FMA, or AVX512 depending on the ISA supported on the target CPU, you can use the configure option `--enable-optimizations` to set the desired optimization flags that will override the default flags.

For example, on a AMD “Zen4”-based processor, you can set 'znver4' flag for improved performance:

```
$ ./configure --enable-amd-flags --enable-optimizations="-march=znver4 -O3"
```

or

```
$ ./configure --enable-amd-flags --enable-optimizations="-march=native -O3"
```

Ensure that the compiler you use supports 'znver4' flag.

### 5.1.2 Using Pre-built Libraries

You can find the AOCL-LAPACK library binaries for Linux at the following URL:

<https://www.amd.com/en/developer/aocl.html#libflame>

Also, the AOCL-LAPACK binary can be installed from the AOCL master installer tar file available at the following URL:

<https://www.amd.com/en/developer/aocl.html>

The tar file includes pre-built binaries of the other AMD libraries as explained in *"Using Master Package" on page 18*.

## 5.2 Usage on Linux

The AOCL-LAPACK source directory contains test cases which demonstrate the usage of AOCL-LAPACK APIs.

From AOCL 3.2, a separate test suite is included for the LAPACK interfaces. Currently, it has test cases for a few AOCL-LAPACK APIs. More test cases will be added in future releases. The test suite validates the APIs and displays performance numbers. The configuration files for input supports testing for a range of input sizes and different parameter values. For more information on this test suite, refer to the *ReadMe.txt* file in the directory *test/main*.

### 5.2.1 Use by Applications

To use AOCL-LAPACK in your application, link with AOCL-LAPACK and AOCL-BLAS library while building the application.

AOCL-LAPACK 4.1 and later have dependency on libstdc++ library. Hence, you must link libstdc++(-lstdc++) when using AOCL-LAPACK library.

An example program demonstrating the usage of AOCL-LAPACK is located at *libflame/test/example*. This directory contains example source file showing the usage of AOCL-LAPACK library functions.

Use the included CMake script to compile and execute the program. You can test it on both Linux and Windows.

1. Move to installed examples directory:

```
$ cd test/example
```

2. Configure the build system:

```
$ mkdir build
$ cd build
$ cmake .. -DEXT_BLAS_LIBRARY_DEPENDENCY_PATH=< path to blas library> -
DEXT_LAPACK_LIBRARY_PATH=<path to AOCL-LAPACK library> -DEXT_BLAS_LIBNAME=blas_lib_name -
DEXT_LAPACK_LIBNAME=lapack_lib_name -DEXT_FLAME_HEADER_PATH=<path to AOCL-LAPACK header file
FLAME.h>
```

Example:

```
$ cmake .. -DEXT_BLAS_LIBRARY_DEPENDENCY_PATH=/home/user/blis -DEXT_LAPACK_LIBRARY_PATH=/home/
user/libflame -DEXT_BLAS_LIBNAME=libblis-mt.a -DEXT_LAPACK_LIBNAME=libflame.a -
DEXT_FLAME_HEADER_PATH=/home/user/aocl/include
```

3. Compile the sample applications:

**For Linux**

```
$ cmake --build . or make
```

**For Windows**

```
$ cmake --build .
```

#### 4. Run the application

```

For Linux

$ ./test_dgetrf.x

For Windows

cd Debug
$ test_dgetrf.exe

```

## 5.3 Building AOCL-LAPACK from Source on Windows

AOCL-LAPACK (<https://github.com/amd/libflame>) uses CMake along with Microsoft Visual Studio for building binaries from the source on Windows. The following sections explain the GUI and command-line schemes of building the binaries and test suite.

### Prerequisites

Refer to the Prerequisites sub-section in *"Build AOCL-BLAS from Source on Windows" on page 49*.

### 5.3.1 Building AOCL-LAPACK Using GUI

#### 5.3.1.1 Preparing Project with CMake GUI

Complete the following steps in the CMake GUI:

1. Set the **source** (folder containing AOCL-LAPACK source code) and **build** (folder in which the project files will be generated, for example, **out**) folder paths. It is not recommended to use the folder named **build** as a folder with that name exists at the top of AOCL-LAPACK source tree.
2. Click on the **Configure** button to prepare the project options.
3. Set the generator to **Visual Studio 17 2022** and the compiler to **ClangCl** or **LLVM**.
4. Update the options based on the project requirements. All the available options are listed in the following table:

**Table 12. AOCL-LAPACK Config Options**

Feature	CMake Parameter(s)
Shared library	BUILD_SHARED_LIBS=ON
Static library	BUILD_SHARED_LIBS=OFF

**Table 12. AOCL-LAPACK Config Options**

Feature	CMake Parameter(s)
Flags enabled by default	BUILD_SHARED_LIBS ENABLE_WINDOWS_BUILD ENABLE_AMD_FLAGS ENABLE_BLAS_EXT_GEMMT ENABLE_MULTITHREADING ENABLE_WRAPPER ENABLE_BLIS1_USE_OF_FLA_MALLOC ENABLE_BUILTIN_LAPACK2FLAME ENABLE_EXT_LAPACK_INTERFACE ENABLE_INTERNAL_ERROR_CHECKING ENABLE_NON_CRITICAL_CODE ENABLE_PORTABLE_TIMER INCLUDE_LAPACKE
Enable AMD optimized path	ENABLE_AMD_OPT=ON <i>Note: It is automatically set to ON when ENABLE_AMD_FLAGS is ON.</i>
32-bit integer size	ENABLE_ILP64=OFF
64-bit integer size	ENABLE_ILP64=ON
BLAS library path	CMAKE_EXT_BLAS_LIBRARY_DEPENDENCY_PATH=<path to BLAS library>
BLAS library name	EXT_BLAS_LIBNAME=BLAS Library Name
Enable invoking 'void' return based interface for BLAS functions DOTC and DOTU	ENABLE_F2C_DOTC=ON
Enable 'void' return type for AOCL-LAPACK functions such as cladiv/zladiv	ENABLE_VOID_RETURN_COMPLEX_FUNCTION=ON
Enables multithreading	ENABLE_MULTITHREADING=ON

**Table 12. AOCL-LAPACK Config Options**

Feature	CMake Parameter(s)
On Windows, setting <code>ENABLE_AMD_FLAGS</code> flag internally enables: <ul style="list-style-type: none"> <li>• <code>ENABLE_BLAS_EXT_GEMMT</code></li> <li>• <code>ENABLE_AMD_OPT</code></li> <li>• <code>ENABLE_BUILTIN_LAPACK2FLAME</code></li> <li>• <code>ENABLE_EXT_LAPACK_INTERFACE</code></li> <li>• <code>ENABLE_F2C_DOTC</code></li> <li>• <code>ENABLE_VOID_RETURN_COMPLEX_FUNCTION</code></li> <li>• <code>ENABLE_MULTITHREADING</code></li> </ul>	<code>ENABLE_AMD_FLAGS=ON</code>
Set external libaoclutils library path	<code>LIBAOCLUTILS_LIBRARY_PATH=&lt;path to libaoclutils library&gt;</code>
Set external libaoclutils header path	<code>LIBAOCLUTILS_INCLUDE_PATH=&lt;path to libaoclutils header files path&gt;</code>
Enable main test suite	<code>BUILD_TEST=ON</code> (ensure that <code>BUILD_LEGACY_TEST</code> is not set)
Enable legacy test suite	<code>BUILD_LEGACY_TEST=ON</code> (ensure that <code>BUILD_TEST</code> is not set)
Set BLAS library header path	<code>BLAS_HEADER_PATH</code> (needed for main test suite)
Enable Netlib test suite	<code>BUILD_NETLIB_TEST=ON</code>

5. Provide the path to the BLAS library. It will be used at the linking stage while building the test suite.

6. To generate the Microsoft Visual Studio project in the **out** folder, click on the **Generate** button as shown in the following figure:

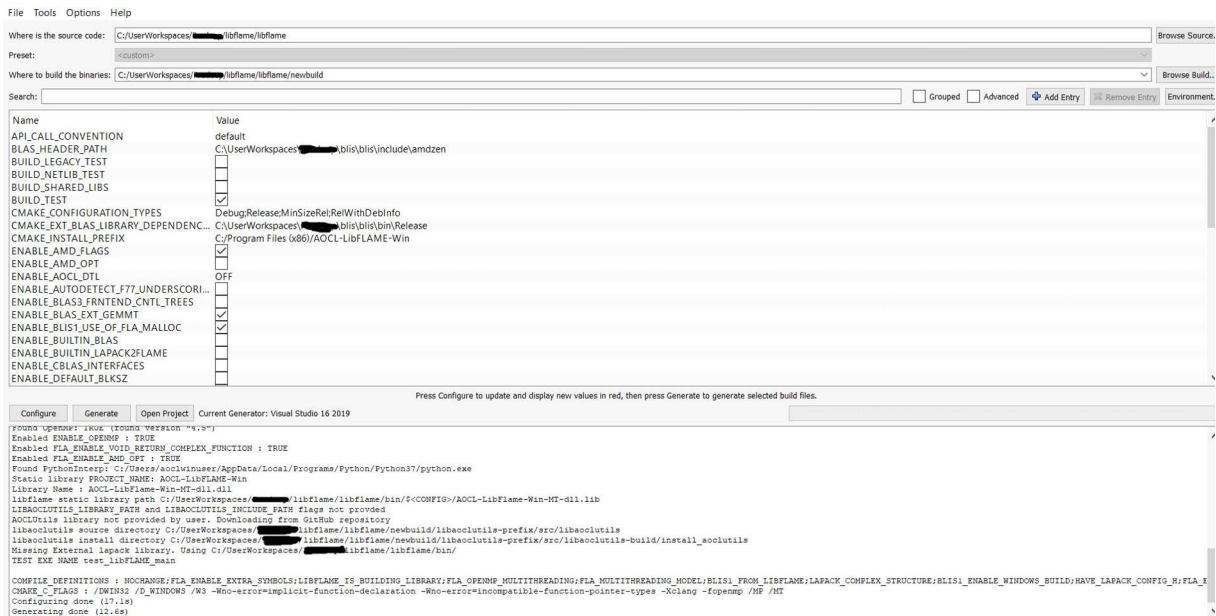


Figure 8. AOCL-LAPACK CMake Configurations

### 5.3.1.2 Building the Project in Visual Studio GUI

Complete the following steps in the Microsoft Visual Studio GUI:

1. Open the project generated by CMake (build folder) in "Preparing Project with CMake GUI" on page 67.
2. To generate AOCL-LAPACK binaries, build the **AOCL-LibFLAME-Win** project.

The library files will generate in the **lib** folder based on the project settings.

For example, *libflame/lib/Release/AOCL-LibFLAME-Win-dll.dll* or *AOCL-LibFLAME-Win-dll.lib*

### 5.3.2 Building AOCL-LAPACK using Command-line Arguments

The project configuration and build procedures can also be triggered from the command prompt. The corresponding steps are described in the following sections.

### 5.3.2.1 Configuring the Project in Command Prompt

In the AOCL-LAPACK project folder, create a folder **out**. Open the command prompt in this directory and run the following command to configure the project:

```
cmake -S .. -B . Example for building ILP64 mode binaries:
cmake -S .. -B . -G "Visual Studio 17 2022" -DCMAKE_BUILD_TYPE=Release -DBUILD_SHARED_LIBS=ON -
DEXT_BLAS_LIBNAME="AOCL-LibBlis-Win-MT-dll.lib" -
DCMAKE_EXT_BLAS_LIBRARY_DEPENDENCY_PATH="<path to AOCL-BLAS library>" -DENABLE_ILP64=ON -
DENABLE_AMD_FLAGS=ON -TLLVM -DBUILD_TEST=OFF -DBUILD_NETLIB_TEST=OFF -DENABLE_WRAPPER=ON -
DOpenMP_libomp_LIBRARY="C:\Program
Files\LLVM\lib\libomp.lib"
```

You can refer to [Table 12](#) and update the parameter options according to the project requirements.

### 5.3.2.2 Building the Project in Command Prompt

Open a command prompt in the `libflame\out` directory. Invoke CMake with the build command with release or debug option. For example:

```
cmake --build . --config Release
```

The library files would be generated in the **Release** or **Debug** folder based on the project settings.

### 5.3.3 Building and Running Test Suite

The Microsoft Visual Studio project for the test suite is generated as a part the CMake generate step. You can build the test projects from the Microsoft Visual Studio GUI or the command prompt as described in the previous sections.

As mentioned in Table 12, enable "BUILD\_TEST" to build a new main test suite of AOCL-LAPACK. To build a legacy test suite, set "BUILD\_LEGACY\_TEST".

**Note:** Both main test suite and legacy test suites must not be enabled together in the same build due to certain incompatible flag settings between the 2 projects.

## 5.4 Checking AOCL-LAPACK Operation Progress

AOCL libraries perform tasks that can be computationally expensive. The AOCL Progress feature provides a mechanism, for a selected set of APIs, for the calling application to check how far a computation has progressed through a callback function.

### Usage

The application must define the `aocl fla_progress` or callback function in a specific format and register this callback function with the AOCL-LAPACK library.

The callback function prototype must be defined as follows:

```
int aocl fla_progress(const char* const api,
const integer lenapi,
const integer* const progress,
const integer* const current_thread,
const integer* const total_threads)
```

However, you can change the function name as per your preference.

The following table explains AOCL-LAPACK Progress feature callback function parameters:

**Table 13. AOCL-LAPACK Progress Feature Callback Function Parameters**

Parameter	Purpose
api	Name of the API running currently
lenapi	Length of the API name character buffer
progress	Linear progress made in the current thread so far
current_thread	Current thread ID
total_threads	Total number of threads used to perform the operation

## Callback Registration

The callback function must be registered with the library to report the progress. Each library has its own callback registration function. The registration is done by calling:

```
aocl fla_set_progress(test_progress);
```

Example:

```
int aocl fla_progress(const char* const api,const integer lenapi,const integer* const
progress,const integer* const current_thread,const integer* const total_threads)
{
    printf( "In AOCL FLA Progress thread %lld", at API %s, progress %lld total threads=
%lld\n",*current_thread, api, *progress,*total_threads );
    return 0;
}
```

or

```
int test_progress(const char* const api,const integer lenapi,const integer * const
progress,const integer *const current_thread,const integer *const total_threads)
{
    printf( "In AOCL Progress thread %lld", at API %s, progress %lld total threads=
%lld\n",*current_thread, api, *progress,*total_threads );
    return 0;
}
```

Register the callback with:

```
aocl fla_set_progress(test_progress);
```



## Limitations

On Windows, `aocl fla_progress` is not supported when using AOCL-LAPACK. Hence, the callback function must be registered through `aocl fla_set_progress`.

---

## Chapter 6 AOCL-FFTW

---

AMD optimized version of Fast Fourier Transform Algorithm (FFTW) is a comprehensive collection of fast C routines for computing the Discrete Fourier Transform (DFT) and various special cases thereof that are optimized for AMD EPYC™ and other AMD “Zen”-based processors. It is an open-source implementation of FFTW. It can compute transforms of real and complex valued arrays of arbitrary size and dimension.

### 6.1 Installing

AOCL-FFTW can be installed from the source or pre-built binaries.

#### 6.1.1 Building AOCL-FFTW from Source on Linux

Complete the following steps to build AOCL-FFTW for AMD EPYC™ processor based on the architecture generation:

1. Download the latest stable release of AOCL-FFTW (<https://github.com/amd/amd-fftw>).
2. Depending on the target system and build environment, you must enable/disable the appropriate configure options. Set PATH and LD\_LIBRARY\_PATH to the MPI installation. In the case of building for AMD Optimized FFTW library with AOCC compiler, you must compile and setup OpenMPI with AOCC compiler.

Complete the following steps to compile it for EPYC™ processors and other AMD “Zen”-based processors:

**Note:** For a complete list of options and their description, type `./configure --help`.

– With GCC (default)

**Double Precision FFTW libraries**

```
$ ./configure --enable-sse2 --enable-avx --enable-avx2 --enable-avx512 --enable-mpi --enable-openmp --enable-shared --enable-amd-opt --enable-amd-mpifft --enable-dynamic-dispatcher --prefix=<your-install-dir>
```

**Single Precision FFTW libraries**

```
$ ./configure --enable-sse2 --enable-avx --enable-avx2 --enable-avx512 --enable-mpi --enable-openmp --enable-shared --enable-single --enable-amd-opt --enable-amd-mpifft --enable-dynamic-dispatcher --prefix=<your-install-dir>
```

**Long double FFTW libraries**

```
$ ./configure --enable-shared --enable-openmp --enable-mpi --enable-long-double --enable-amd-opt --enable-amd-mpifft --enable-dynamic-dispatcher --prefix=<your-install-dir>
```

**Quad Precision FFTW libraries**

```
$ ./configure --enable-shared --enable-openmp --enable-quad-precision --enable-amd-opt --enable-dynamic-dispatcher --prefix=<your-install-dir>
```

– With AOCC

**Double Precision FFTW libraries**

```
$ ./configure --enable-sse2 --enable-avx --enable-avx2 --enable-avx512 --enable-mpi --enable-openmp --enable-shared --enable-amd-opt --enable-amd-mpifft --enable-dynamic-dispatcher --prefix=<your-install-dir> CC=clang F77=flang FC=flang
```

**Single Precision FFTW libraries**

```
$ ./configure --enable-sse2 --enable-avx --enable-avx2 --enable-avx512 --enable-mpi --enable-openmp --enable-shared --enable-single --enable-amd-opt --enable-amd-mpifft --enable-dynamic-dispatcher --prefix=<your-install-dir> CC=clang F77=flang FC=flang
```

**Long double FFTW libraries**

```
$ ./configure --enable-shared --enable-openmp --enable-mpi --enable-long-double --enable-amd-opt --enable-amd-mpifft --enable-dynamic-dispatcher --prefix=<your-install-dir> CC=clang F77=flang FC=flang
```

**Quad FFTW libraries**

```
$ ./configure --enable-shared --enable-openmp --enable-quad-precision --enable-amd-opt --enable-dynamic-dispatcher --prefix=<your-install-dir> CC=clang F77=flang FC=flang
```

AMD optimized fast planner is added as an extension to the original planner to improve the planning time of various planning modes in general and PATIENT mode in particular.

The configure user option `--enable-amd-fast-planner` when given in addition to `--enable-amd-opt` enables this new fast planner.

An optional configure option `AMD_ARCH` is supported, that can be set to the CPU architecture values, such as `auto`, `znver1`, `znver2`, `znver3`, or `znver4` for AMD EPYC™ and other AMD “Zen”-based processors.

Additional config and build options to enable specific optimizations are covered in the section “AOCL-FFTW Tuning Guidelines” on page 162.

A dynamic dispatcher feature has been added to build a single portable optimized library for execution on a wide range of x86 CPU architectures. Use the `--enable-dynamic-dispatcher` configure option to enable this feature on Linux-based systems. The configure option `--enable-amd-opt` is the mandatory master optimization switch that must be set for enabling other optional configure options, such as:

- `--enable-amd-mpifft`
- `--enable-amd-mpi-vader-limit`
- `--enable-amd-trans`
- `--enable-amd-fast-planner`
- `--enable-amd-top-n-planner`
- `--enable-amd-app-opt`
- `--enable-dynamic-dispatcher`

### 3. Build the library:

```
$ make
```

### 4. Install the library in the preferred path:

```
$ make install
```

### 5. Verify the installed library:

```
$ make check
```

## 6.1.2 Building AOCL-FFTW from Source on Windows

AOCL-FFTW uses CMake along with Microsoft Visual Studio for building binaries from the sources on Windows. This section explains the GUI and command-line schemes for building the binaries and test suite.

### Prerequisites

The following prerequisites must be met:

- Windows 10/11 and Windows Server 2019/2022
- A suitable MPI library installation along with the appropriate environment variables on the host machine
- LLVM 13/14 for AMD “Zen3” support
- LLVM plug-in for Microsoft Visual Studio (if latest version of LLVM is installed separately, this plugin enables linking Visual Studio with the installed LLVM tool-chain)

- CMake versions 3.0 through 3.23.3
- MPI compiler
- Microsoft Visual Studio 2019 build 16.8.7
- Microsoft Visual Studio tools
  - Python development
  - Desktop development with C++: C++ Clang-Cl for build tool (x64 or x86)

### 6.1.2.1 Using CMake GUI to Build

Complete the following steps in the CMake GUI:

1. Set the **source** (folder containing FFTW source code) and **build** (folder in which the project files will be generated, for example, **out**) folder paths.
2. Click on the **Configure** button to prepare the project options.
3. Set the generator to **Visual Studio 16 2019** or **Visual Studio 17 2022** and the compiler to **ClangCl** or **LLVM**.
4. Update the options based on the project requirements. All the available options are listed in the following table:

**Table 14. AOCL-FFTW Config Options**

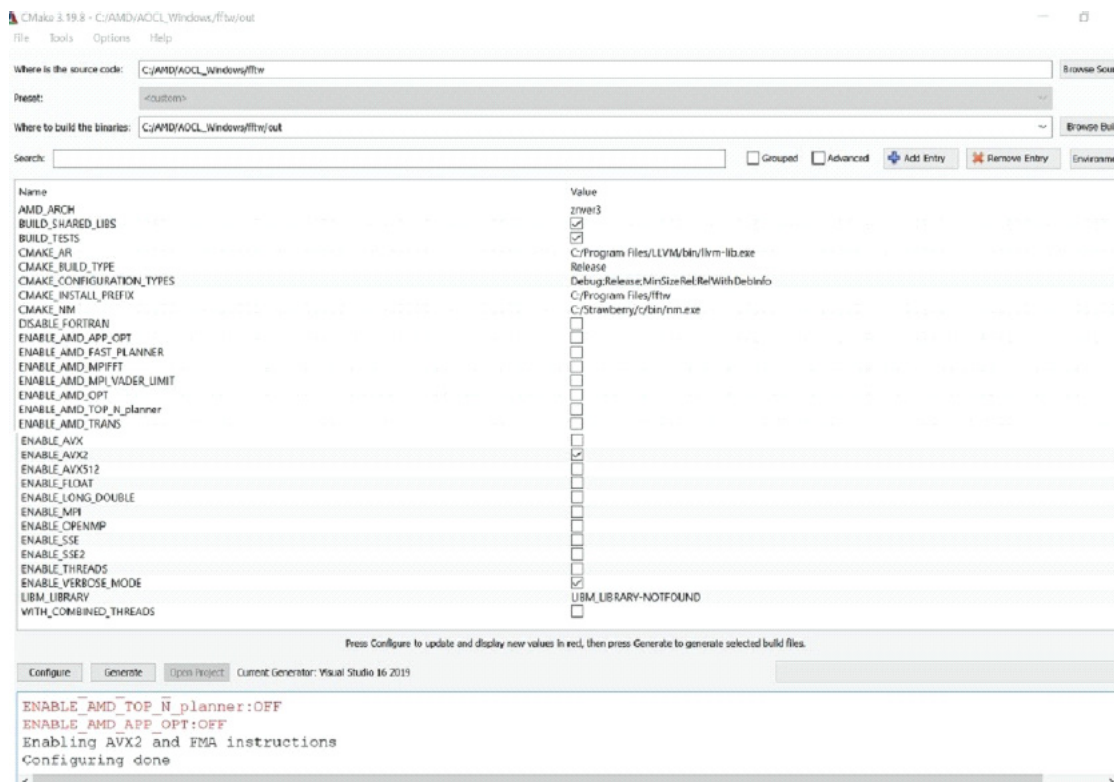
Feature	CMake Parameters
Build type (Release or Debug mode)	CMAKE_BUILD_TYPE=Release/Debug
AMD CPU architecture (AMD “Zen”/AMD “Zen2”/AMD “Zen3”/AMD “Zen4”)	AMD_ARCH: STRING=znver1/znver2/znver3/znver4
Shared library without multithreading	BUILD_SHARED_LIBS=ON ENABLE_OPENMP=OFF ENABLE_THREADS=OFF
Shared library with multithreading	BUILD_SHARED_LIBS=ON ENABLE_OPENMP=ON
Static library without multithreading	BUILD_SHARED_LIBS=OFF ENABLE_OPENMP=OFF
Static library with multithreading	BUILD_SHARED_LIBS=OFF ENABLE_OPENMP=ON
Use Threads instead of OpenMP for multithreading	ENABLE_THREADS=ON WITH_COMBINED_THREADS=ON
Use both Threads and OpenMP for multithreading	ENABLE_THREADS=ON ENABLE_OPENMP=ON

**Table 14. AOCL-FFTW Config Options**

Feature	CMake Parameters
Flags for enhanced instruction set support	ENABLE_SSE=ON ENABLE_SSE2=ON ENABLE_AVX=ON ENABLE_AVX2=ON ENABLE_AVX512=ON
Flags for single and long double	ENABLE_FLOAT=ON ENABLE_LONG_DOUBLE=ON
Build tests directory and generate test applications	BUILD_TESTS=ON
Enables MPI lib	ENABLE_MPI=ON
Enables AMD optimizations	ENABLE_AMD_OPT=ON
Enables AMD MPI FFT optimizations	ENABLE_AMD_MPIFFT=ON ENABLE_AMD_MPI_VADER_LIMIT: ON
Enables AMD optimized transpose	ENABLE_AMD_TRANS=ON
Enables AMD optimizations for HPC/Scientific applications	ENABLE_AMD_APP_OPT: ON

**Note:** *ENABLE\_QUAD\_PRECISION* is currently not supported on Windows.

Select the available and recommended options as follows:



**Figure 9. AOCL-FFTW CMake Config Options**

5. Click the **Generate** button and then **Open Project**.

### 6.1.2.2 Using Command-line Arguments to Build

Complete the following steps to trigger the project configuration and build procedures from the command prompt:

1. In the AOCL-FFTW project folder, create a folder **out**. Open the command prompt in this directory and run the following command to configure the project:

```
cmake .. -DBUILD_TESTS=ON -D[other options1] -D[other options2] -T ClangCl -G "Visual Studio 16 2019" && cmake --build . --config Release
```

2. Refer Table 14 and update the parameter options in the command according to the project requirements.

The library files would be generated in the **Release** or **Debug** folder based on the project settings.

3. To verify the installed library, copy the test scripts from `\win\tests` to `\out\Release` and run `python fftw_check.py`.

### 6.1.3 Using Pre-built Libraries

The AOCL-FFTW library binaries for Linux and Windows are available at the following URL:

<https://www.amd.com/en/developer/aocl/fftw.html>

The AOCL-FFTW binary for Linux and Windows can also be installed from the AOCL master installer (tar packages for Linux and zip packages for Windows) available at the following URL:

<https://www.amd.com/en/developer/aocl.html>

The *tar* and *zip* files include pre-built binaries of other AMD libraries as explained in “Using Master Package” on page 18.

**Note:** *The pre-built libraries are prepared on a specific platform having dependencies related to OS, Compiler (GCC, Clang), MPI, Visual studio, and GLIBC. Your platform must adhere to the same versions of these dependencies to use the pre-built libraries.*

## 6.2 Usage

Sample programs and executable binaries demonstrating the usage of AOCL-FFTW APIs and performance benchmarking are available in *tests/* and *mpi/* directories for Linux and *out/Release* directory for Windows.

### 6.2.1 Sample Programs for Single-threaded and Multi-threaded FFTW

To run single-threaded test, execute the following command:

```
$ bench -opatient -s [i|o][r|c][f|b]<size>
```

Where,

- i/o means in-place or out-of-place. Out of place is the default.
- r/c means real or complex transform. Complex is the default.
- f/b means forward or backward transform. Forward is the default.
- <size> is an arbitrary multidimensional sequence of integers separated by the character 'x'.

Check the tuning guidelines for single-threaded test execution in “AOCL-FFTW Tuning Guidelines” on page 162.

To run multi-threaded test, execute the following command:

```
$bench -opatient -onthreads=N -s [i|o][r|c][f|b]<size>
```

Where, N is number of threads.

Check the tuning guidelines for multi-threaded test execution in the section “AOCL-FFTW Tuning Guidelines” on page 162.



## 6.2.2 Sample Programs for MPI FFTW

```
$mpirun -np N mpi-bench -opatient -s [i|o][r|c][f|b]<size>
```

Where, N is the number of processes.

Check the tuning guidelines for MPI test execution in the section “AOCL-FFTW Tuning Guidelines” on page 162.

## 6.2.3 Additional Options

- `-owisdom`

On startup, read wisdom from the file *wis.dat* in the current directory (if it exists).

On completion, write accumulated wisdom to *wis.dat* (overwriting if file exists).

This bypasses the planner next time onwards and directly executes the read plan from wisdom.

- `--verify <problem>`

Verify that AOCL-FFTW is computing correctly. It does not output anything unless there is an error.

- `-v<n>`

Set verbosity to <n> or 1 if <n> is omitted. -v2 will output the created plans.

### Notes:

1. The names of windows FFTW test bench application has .exe extension (*bench.exe* and *mpi-bench.exe*).
2. The folder */win/tests/* includes Windows benchmark scripts for single-threaded, multi-threaded and MPI FFT execution for standard sizes. A *README* file is also provided with the instructions to run these benchmark scripts.

To display the AOCL version number of AOCL-FFTW library, application must call the following FFTW API *fftw\_aoclversion()*.

The test bench executables of AOCL-FFTW support the display of AOCL version using the `--info-all` option.

## Chapter 7 AOCL-LibM

AOCL-LibM is a high-performant implementation of LibM, the standard C library of basic floating-point mathematical functions. It includes many of the functions from the C99 standard. Single and double precision versions of the functions are provided, all optimized for accuracy and performance, including a small number of complex functions. There are also a number of vector and fast scalar variants provided, in which a small amount of the accuracy has been traded for greater performance.

### 7.1 Library Contents

A list of the scalar functions present in the library is provided below.

**Note:** An “f” at the end of the function name indicates that it is single-precision; otherwise, it is double-precision. They can be called by a standard C99 function and naming convention and must be linked with AOCL-LibM before standard libm.

For example:

```
$ export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/path/to/AOCL-LibM_library
$ clang -Wall -std=c99 myprogram.c -o myprogram -L<Path to AOCL-LibM Library> -lalm -lm

Or

$ gcc -Wall -std=c99 myprogram.c -o myprogram -L<Path to AOCL-LibM Library> -lalm -lm
```

- Trigonometric
  - cosf, cos, sinf, sin, tanf, tan, sincosf, and sincos
- Inverse Trigonometric
  - acosf, acos, asinf, asin, atanf, atan, atan2f, and atan2
- Hyperbolic
  - coshf, cosh, sinh, tanhf, and tanh
- Inverse Hyperbolic
  - acoshf, acosh, asinhf, asinh, atanhf, and atanh
- Exponential and Logarithmic
  - expf, exp, exp2f, exp2, exp10f, exp10, expm1f, and expm1
  - logf, log, log10f, log10, log2f, log2, log1pf, and log1p
  - logbf, logb, ilogbf, and ilogb
  - modff, modf, frexpf, frexp, ldexpf, and ldexp
  - scalbnf, scalbn, scalblnf, and scalbln

- Error
  - erff and erf
- Power and Absolute Value
  - powf, pow, fastpow, cbrtf, cbrt, sqrtf, sqrt, hypotf, and hypot
  - fabsf and fabs
- Nearest Integer
  - ceilf, ceil, floorf, floor, truncf, and trunc
  - rintf, rint, roundf, round, nearbyintf, and nearbyint
  - lrintf, lrint, llrintf, and llrint
  - lroundf, lround, llroundf, and llround
- Remainder
  - fmodf, fmod, remainderf, and remainder
- Manipulation
  - copysignf, copysign, nanf, nan, finitf, and finite
  - nextafterf, nextafter, nexttowardf, and nexttoward
- Maximum, Minimum, and Difference
  - fdimf, fdim, fmaxf, fmax, fminf, and fmin

A fast version of AOCL-LibM is available in the library *libalmfast.so*. This library contains faster variants of the scalar functions `acos`, `asin`, `asinf`, `atan`, `atanf`, `erf`, `erff`, `exp`, `expf`, `log`, `logf`, `powf`, `tan`, and `tanf`. These functions can be accessed by directly linking to this library before *libalm.so*, can be selected by setting `LD_PRELOAD=/path-to/libalmfast.so` or enabled through the use of certain flags by the AOCC compiler. For more information, refer to the AOCC 4.1 user guide.

AOCL-LibM includes the vector variants for the core math functions: power, exponential, logarithmic, and trigonometric. A few caveats on the vector variants are as follows:

- The vector variants trade off some of the accuracy for increased performance, but should nevertheless have a maximum ULP error no greater than 4.0.
- While these routines take advantage of the AMD64 architecture for performance, some improvement is also made by sacrificing error handling and argument checking.
- Abnormal inputs may produce unpredictable results. It is therefore the responsibility of the caller of these routines to ensure that their arguments are valid.
- The vector variants do not set the IEEE error codes and hence, the user code must not rely on them.
- The vector routines must be invoked using the C intrinsics or from the x86 assembly.

The vector variants can be enabled by using the AOCC compiler with the `-ffast-math` flag. You can also call these functions directly; if doing so, you must take care to avoid losing portability. As these

functions accept arguments in `__m128`, `__m128d`, `__m256`, `__m256d`, `__m512` and `__m512d` types, you must manually pack and later unpack to and from the appropriate data type.

The following naming convention is used:

```
amd_vr<type><vec_size>_<func>
```

where,

- v – vector
- r – real
- a – array
- <type> - ‘s’ for single precision and ‘d’ for double precision
- <vec\_size> - 4, 8, or 16 for single-precision; 2, 4, or 8 for double-precision
- <func> - function name, such as ‘exp’ and ‘expf’

For example, a single precision 4 element version of exp has the signature:

```
__m128 amd_vrs4_expf (__m128 x);
```

The list of available vector functions is as follows:

**Note:** All these functions have an ‘amd\_’ prefix, but this has been omitted in the following list for brevity.

- Exponential
  - vrs8\_expf and vrs8\_exp2f
  - vrs4\_expf, vrs4\_exp2f, vrs4\_exp10f, and vrs4\_expm1f
  - vrsa\_expf, vrsa\_exp2f, vrsa\_exp10f, and vrsa\_expm1f
  - vrd2\_exp, vrd2\_exp2, vrd2\_exp10, vrd2\_expm1, vrd4\_exp, and vrd4\_exp2
  - vrda\_exp, vrda\_exp2, vrda\_exp10, and vrda\_expm1
  - vrs16\_expf and vrs16\_exp2f
  - vrd8\_exp and vrd8\_exp2
- Logarithmic
  - vrs8\_logf, vrs8\_log2f, and vrs8\_log10f
  - vrs4\_logf, vrs4\_log2f, vrs4\_log10f, and vrs4\_log1pf
  - vrd4\_log and vrd4\_log2
  - vrsa\_logf, vrsa\_log2f, vrsa\_log10f, and vrsa\_log1pf
  - vrd2\_log, vrd2\_log2, vrd2\_log10, and vrd2\_log1p
  - vrda\_log, vrda\_log2, vrda\_log10, vrda\_log1p
  - vrs16\_logf, vrs16\_log2f, and vrs16\_log10f
  - vrd8\_log and vrd8\_log2

- Trigonometric
  - vrs4\_coshf, vrs8\_coshf, vrs4\_sinhf, and vrs8\_sinhf
  - vrsa\_coshf, vrsa\_sinhf, and vrsa\_sincoshf
  - vrd4\_sin, vrd4\_cos, vrd4\_tan, and vrd4\_sincos
  - vrd2\_cos, vrd2\_sin, vrd2\_tan, and vrd2\_sincos
  - vrda\_cos, vrda\_sin, and vrda\_sincos
  - vrs16\_coshf, vrs16\_sinhf, and vrs16\_tanhf
  - vrd8\_cos, vrd8\_sin, vrd8\_tan, and vrd8\_sincos
- Inverse Trigonometric
  - vrs4\_acoshf, vrs4\_asinhf, and vrs8\_asinhf
  - vrs4\_atanhf, vrs8\_atanhf, and vrd2\_atan
  - vrs16\_atanhf, vrs16\_asinhf, and vrs16\_acoshf
  - vrd8\_atan and vrd8\_asin
- Hyperbolic
  - vrs4\_coshf and vrs4\_tanhf
  - vrs8\_coshf and vrs8\_tanhf
  - vrs16\_tanhf
- Power
  - vrs4\_powf, vrd2\_pow, vrd4\_pow, vrs8\_powf, and vrsa\_powf
  - vrs16\_powf and vrd8\_pow
- Error
  - vrs4\_erff, vrd2\_erf, vrs8\_erff, and vrd4\_erf
  - vrd16\_erff and vrd8\_erf
- Vector Array Arithmetic Functions
  - vrsa\_addf, vrsa\_addfi, vrda\_add, and vrda\_addi
  - vrsa\_subf, vrsa\_subfi, vrda\_sub, and vrda\_subi
  - vrsa\_mulf, vrsa\_mulfi, vrda\_mul, and vrda\_muli
  - vrsa\_divf, vrsa\_divfi, vrda\_div, and vrda\_divi

## 7.2 Installation

### 7.2.1 Installing the Pre-Built Binaries on Linux

The AOCL-LibM binary for Linux is available at the following URL:

<https://www.amd.com/en/developer/aocl/libm.html>

The AOCL-LibM library can also be installed from the AOCC and GCC compiled AOCL master installer tar files available on AMD Developer Central (<https://www.amd.com/en/developer/aocl.html>).

The *tar* and *zip* files include pre-built binaries of other AOCL libraries as explained in [Using Master Package](#).

## 7.2.2 Building AOCL-LibM on Linux

Software requirements for compilation:

- GCC versions 9.2 through v13.1
- Glibc versions 2.29 through v2.31
- Clang 12.0.0 (AOCC 3.0) through Clang 14.0.0 (AOCC 4.0)
- Virtualenv with Python 3.6 or later
- SCons versions 3.1.1 or later
- libstdc++ (required for AOCL-Utils)

Refer to [Chapter 3](#) to install the AOCL-Utils library. Then, complete the following steps to compile AOCL-LibM:

1. Download source from GitHub (<https://github.com/amd/aocl-libm-ose>).
2. Navigate to the LibM folder and checkout to the branch *aocl-4.1*:

```
cd aocl-libm-ose
git checkout aocl-4.1
```

3. Create a virtual environment:

```
virtualenv -p python3 .venv3
```

4. Activate the virtual environment:

```
source .venv3/bin/activate
```

5. Install SCons:

```
pip install scons
```

6. Compile AOCL-LibM:

```
Basic build command: scons --aocl_utils_install_path=<libaoclutils library path>
```

Additional Flags

```
Build in parallel: -j<number of parallel builds>
```

```
Installation: install --prefix=<path to install>
```

```
Compiler selection: ALM_CC=<gcc/clang executable path> ALM_CXX=<g++/clang++ executable path>
```

```
Verbosity: --verbose=1
```

```
Debug mode build: --debug_mode=libs
```

7. The libraries (static and dynamic) will be compiled and generated in the following location:

*aocl-libm-ose/build/aocl-release/src/*

If the above installation option is used, the libraries will also be copied to the directory *<path to install>/lib*.

### 7.2.3 Building AOCL-LibM on Windows

Minimum software requirements for compilation:

- Windows 10/11 or Windows Server 2019/2022
- LLVM compiler V14.0 for AMD “Zen3” or AMD “Zen4” support (or LLVM compiler V11.0 for AMD “Zen2” support)
- Microsoft Visual Studio 2019 build 16 or 2022 build 17
- Windows SDK Version 10.0.19041.0
- Virtualenv with Python 3.6 or later
- SCons 4.4.0
- libstdc++ (required for AOCL-Utils)

Refer to [Chapter 3](#) to install the AOCL-Utils library. Then, complete the following steps to install AOCL-LibM:

1. Download source from GitHub (<https://github.com/amd/aocl-libm-ose>).
2. Navigate to the folder:

```
cd aocl-libm-ose
```

3. Install virtualenv:

```
pip install virtualenv
```

4. Initialize the environment for correct architecture using Visual Studio vcvarsall.bat file using following command:

```
"C:\Program Files (x86)\Microsoft Visual  
Studio\2019\Community\VC\Auxiliary\Build\vcvarsall.bat" amd64
```

5. Activate virtual environment and install SCons inside:

```
virtualenv -p python .venv3  
.venv3\Scripts\activate  
pip install scons
```

## 6. Build the project using clang compiler:

```
Basic build command: scons ALM_CC=<clang-cl executable path> ALM_CXX=<clang-cl executable path>
--aocl_utils_install_path=<libaoclutils library path>
```

Additional Flags

Build in parallel: -j<number of parallel builds>

Verbosity: --verbose=1

Debug mode build: --debug\_mode=libs

For example:

```
scons -j32 ALM_CC="C:\PROGRA~1\LLVM\bin\clang-cl.exe" ALM_CXX="C:\PROGRA~1\LLVM\bin\clang-cl.exe" --verbose=1
```

The static (*libalm-static.lib*) and dynamic (*libalm.dll* and *libalm.lib*) libraries are compiled and generated in the following location:

*aocl-libm-ose/build/aocl-release/src/*

## 7.3 Using AOCL-LibM

To use AOCL-LibM in your application, complete the following steps:

1. Include 'math.h' as a standard way to use the C Standard library math functions.
2. Link in the appropriate version of the library in your program.

The Linux libraries may have a dependency on the system math library. When linking AOCL-LibM, ensure that it precedes the system math library in the link order, that is, *-lalm* must appear before *-lm*. The explicit linking of the system math library is required when using the GCC or AOCC compilers. Such explicit linking is not required with the g++ compiler (for C++).

Example: myprogram.c

```
#include <stdio.h>
#include <math.h>

int main() {
    float f = 3.14f;
    printf ("%f\n", expf(f));
    return 0;
}
```

To use AOCL-LibM scalar functions, use the following commands:

```
$ export LD_LIBRARY_PATH=<Path to libalm.so>:$LD_LIBRARY_PATH
$ cc -Wall -std=c99 myprogram.c -o myprogram -L<Path to libalm.so> -lalm -lm (cc can be 'gcc' or
'clang').
$ ./myprogram
```

You can use the vector calls by adding the compiler flag *-ffast-math*.



You can also call the functions directly, which requires manual packing and unpacking. To do so, you must include the header file *amdlibm\_vec.h*. The following program shows such an example. For simplicity, the size and other checks are omitted.

Example: *myprogram.c*

```
##define AMD_LIBM_VEC_EXTERNAL_H
#define AMD_LIBM_VEC_EXPERIMENTAL
#include "amdlibm_vec.h"
__m128 vrs4_expf (__m128 x);

__m128
test_expf_v4s(float *ip, float *out)
{
    __m128 ip4 = _mm_set_ps(ip[3], ip[2], ip[1], ip[0]);
    __m128 op4 = vrs4_expf(ip4);
    _mm_store_ps(&out[0], op4);

    return op4;
}
```

You can compile *myprogram.c* as follows:

```
$ export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/path/to/AOCL-LibM
$ clang -Wall -std=c99 -ffastmath myprogram.c -o myprogram -L<path to libalm.so> -lalm -lm
```

For more details on usage, refer to the examples folder in the release package, which contains example source code and a makefile.

## Chapter 8 AOCL-ScaLAPACK

---

AOCL-ScaLAPACK is a library of high-performance linear algebra routines for parallel distributed memory machines. It can be used to solve linear systems, least squares problems, eigenvalue problems, and singular value problems. AOCL-ScaLAPACK is optimized for AMD “Zen”-based processors. It depends on the external libraries BLAS and LAPACK; thus, the use of AOCL-BLAS and AOCL-LAPACK is recommended.

### 8.1 Installation

AOCL-ScaLAPACK can be installed from source or pre-built binaries.

#### 8.1.1 Building AOCL-ScaLAPACK from Source on Linux

GitHub URL: <https://github.com/amd/aocl-scalapack>

##### Prerequisites

Building AOCL-ScaLAPACK library requires linking to the following libraries installed using pre-built binaries or built from source:

- AOCL-BLAS
- AOCL-LAPACK
- AOCL-Utills
- An MPI library (validated with OpenMPI library)

Complete the following steps to build AOCL-ScaLAPACK from source:

1. Clone the GitHub repository (<https://github.com/amd/aocl-scalapack.git>).
2. Execute the command:

```
$ cd scalapack
```

3. CMake as follows:

- a. Create a new directory. For example, build:

```
$ mkdir build  
$ cd build
```

- b. Set PATH and LD\_LIBRARY\_PATH appropriately to the MPI installation.
- c. Run cmake command based on the compiler and the type of library generation required.

**Note:** AOCL-LAPACK 4.1 version and later has dependency on libstdc++ library. Hence, user must link libstdc++(-lstdc++) while specifying the path for LAPACK\_LIBRARIES in the CMake flags.

**Table 15. Compiler and Type of Library**

Compiler	Library Type	Threading	Command [<>] - use if ILP64 binary required
GCC	Static	Single-thread AOCL-BLAS	\$ cmake .. -DBUILD_SHARED_LIBS=OFF -DBLAS_LIBRARIES="-fopenmp <path to AOCL-BLAS library>/libblis.a" -DLAPACK_LIBRARIES="-lstdc++ <path to AOCL-LAPACK library>/libflame.a" -DCMAKE_C_COMPILER=mpicc -DCMAKE_Fortran_COMPILER=mpif90 -DUSE_OPTIMIZED_LAPACK_BLAS=OFF [-D DENABLE_ILP64=ON]
		Multi-thread AOCL-BLAS	\$ cmake .. -DBUILD_SHARED_LIBS=OFF -DBLAS_LIBRARIES="-fopenmp <path to AOCL-BLAS library>/libblis-mt.a" -DLAPACK_LIBRARIES="-lstdc++ <path to AOCL-LAPACK library>/libflame.a" -DCMAKE_C_COMPILER=mpicc -DCMAKE_Fortran_COMPILER=mpif90 -DUSE_OPTIMIZED_LAPACK_BLAS=OFF [-D DENABLE_ILP64=ON]
	Shared	Single-thread AOCL-BLAS	\$ cmake .. -DBUILD_SHARED_LIBS=ON -DBLAS_LIBRARIES="-fopenmp <path to AOCL-BLAS library>/libblis.so" -DLAPACK_LIBRARIES="-lstdc++ <path to AOCL-Utils library>/libaoclutils.so <path to AOCL-LAPACK library>/libflame.so" -DCMAKE_C_COMPILER=mpicc -DCMAKE_Fortran_COMPILER=mpif90 -DUSE_OPTIMIZED_LAPACK_BLAS=OFF [-D DENABLE_ILP64=ON]
		Multi-thread AOCL-BLAS	\$ cmake .. -DBUILD_SHARED_LIBS=ON -DBLAS_LIBRARIES="-fopenmp <path to AOCL-BLAS library>/libblis-mt.so" -DLAPACK_LIBRARIES="-lstdc++ <path to AOCL-Utils library>/aoclutils.so <path to AOCL-LAPACK library>/libflame.so" -DCMAKE_C_COMPILER=mpicc -DCMAKE_Fortran_COMPILER=mpif90 -DUSE_OPTIMIZED_LAPACK_BLAS=OFF [-D DENABLE_ILP64=ON]

**Table 15. Compiler and Type of Library**

Compiler	Library Type	Threading	Command [<>] - use if ILP64 binary required
AOCC	Static	Single-thread AOCL-BLAS	\$ cmake .. -DBUILD_SHARED_LIBS=OFF -DBLAS_LIBRARIES="-fopenmp <path to AOCL-BLAS library>/libblis.a" -DLAPACK_LIBRARIES="-lstdc++ <path to AOCL-LAPACK library>/libflame.a" -DCMAKE_C_COMPILER=mpicc -DCMAKE_Fortran_COMPILER=mpif90 -DUSE_OPTIMIZED_LAPACK_BLAS=OFF [-D DENABLE_ILP64=ON]
		Multi-thread AOCL-BLAS	\$ cmake .. -DBUILD_SHARED_LIBS=OFF -DBLAS_LIBRARIES="-fopenmp <path to AOCL-BLAS library>/libblis-mt.a" -DLAPACK_LIBRARIES="-lstdc++ <path to AOCL-LAPACK library>/libflame.a" -DCMAKE_C_COMPILER=mpicc -DCMAKE_Fortran_COMPILER=mpif90 -DUSE_OPTIMIZED_LAPACK_BLAS=OFF [-D DENABLE_ILP64=ON]
	Shared	Single-thread AOCL-BLAS	\$ cmake .. -DBUILD_SHARED_LIBS=ON -DBLAS_LIBRARIES="-fopenmp <path to AOCL-BLAS library>/libblis.so" -DLAPACK_LIBRARIES="-lstdc++ <path to aocc built AOCL-Utills library>/libaoclutils.so <path to AOCL-LAPACK library>/libflame.so" -DCMAKE_C_COMPILER=mpicc -DCMAKE_Fortran_COMPILER=mpif90 -DUSE_OPTIMIZED_LAPACK_BLAS=OFF [-D DENABLE_ILP64=ON]
		Multi-thread AOCL-BLAS	\$ cmake .. -DBUILD_SHARED_LIBS=ON -DBLAS_LIBRARIES="-fopenmp <path to AOCL-BLAS library>/libblis-mt.so" -DLAPACK_LIBRARIES="-lstdc++ <path to aocc built AOCL_utils library>/libaoclutils.so <path to AOCL-LAPACK library>/libflame.so" -DCMAKE_C_COMPILER=mpicc -DCMAKE_Fortran_COMPILER=mpif90 -DUSE_OPTIMIZED_LAPACK_BLAS=OFF [-D DENABLE_ILP64=ON]

On Linux, the inbuilt communications sub-module of ScaLAPACK, called Basic Linear Algebra Communication Subprograms (BLACS), exposes the API symbols in lower case with underscore format.

You can build AOCL-ScaLAPACK with an external BLACS library on Linux using the following configure option:

**Example:** To build static library with external BLACS library:

```
$ cmake .. -DBUILD_SHARED_LIBS=OFF -DBLAS_LIBRARIES="-fopenmp <path to AOCL-BLAS library>/libblis-mt.a" -DLAPACK_LIBRARIES="-lstdc++ <path to AOCL-LAPACK library>/libflame.a" -DBLACS_LIBRARIES=<path to BLACS library>/libBLACS.a -DCMAKE_C_COMPILER=mpicc -DCMAKE_Fortran_COMPILER=mpif90 -DUSE_OPTIMIZED_LAPACK_BLAS=OFF
```

You can build AOCL-ScaLAPACK with Intel MPI and ICC compiler tool chain using the following configure option.

**Example:** To build a static library with Intel MPI and ICC compiler:

```
cmake .. -DBUILD_SHARED_LIBS=OFF -DBLAS_LIBRARIES="-fopenmp <path to AOCL-BLAS library>/libblis-mt.a" -DLAPACK_LIBRARIES="-lstdc++ -fopenmp <path to AOCL-LAPACK library>/libflame.a" -DCMAKE_C_COMPILER=mpiicc -DCMAKE_Fortran_COMPILER=mpiifort -DUSE_OPTIMIZED_LAPACK_BLAS=OFF;
```

- d. Ensure CMake locates AOCL-LAPACK and AOCL-BLAS libraries. On completion, a message, “**LAPACK routine dgesv is found: 1**” similar to the following in CMake output is displayed:

```
...
--
-- CHECKING BLAS AND LAPACK LIBRARIES
-- --> LAPACK supplied by user is <path>/libflame.a.
-- --> LAPACK routine dgesv is found: 1.
-- --> LAPACK supplied by user is WORKING, will use <path>/libflame.a.
-- BLAS library: <path>/libblis.a
-- LAPACK library: <path>/libflame.a
...
...
```

- e. Compile the code:

```
$ make -j
```

When the building process is complete, the AOCL-ScaLAPACK library is created in the lib directory. The test application binaries are generated in the `<aocl-scalapack>/build/TESTING` folder.

## 8.1.2 Using Pre-built Libraries

AOCL-ScaLAPACK library binaries for Linux are available at the following URL:

<https://www.amd.com/en/developer/aocl/scalapack.html>

Also, AOCL-ScaLAPACK binary can be installed from the AOCL master installer tar file available at the following URL:

<https://www.amd.com/en/developer/aocl.html>

The tar file includes pre-built binaries of other AMD Libraries as explained in “*Using Master Package*” on page 18.

## 8.2 Usage

You can find the applications demonstrating the usage of ScaLAPACK APIs in the TESTING directory of ScaLAPACK source package:

```
$ cd scalapack/TESTING
```

## 8.3 Building AOCL-ScaLAPACK from Source on Windows

GitHub URL: <https://github.com/amd/aocl-scalapack>

AOCL-ScaLAPACK uses CMake along with Microsoft Visual Studio for building the binaries from the sources on Windows. The following sections explain the GUI and command-line schemes of building the binaries and test suite.

### Prerequisites

The following prerequisites must be met:

- AOCL-BLAS, AOCL-LAPACK, and AOCL-Utils libraries
- Windows 10/11 or Windows Server 2019/2022
- LLVM 15/16
- LLVM plug-in for Microsoft Visual Studio (if latest version of LLVM is installed separately, this plug-in enables linking Microsoft Visual Studio with the installed LLVM tool-chain)
- CMake versions 3.0 through 3.23.3
- MPI compiler
- Fortran 90 compiler
- Microsoft Visual Studio 2019 (build 16.8.7) through 2022 (build 17.3.2)
- Microsoft Visual Studio tools
  - Python development
  - Desktop development with C++: C++ Clang-Cl for v142 build tool (x64 or x86)

### 8.3.1 Building AOCL-ScaLAPACK Using GUI

#### 8.3.1.1 Preparing Project with CMake GUI

Complete the following steps to prepare the project with CMake GUI:

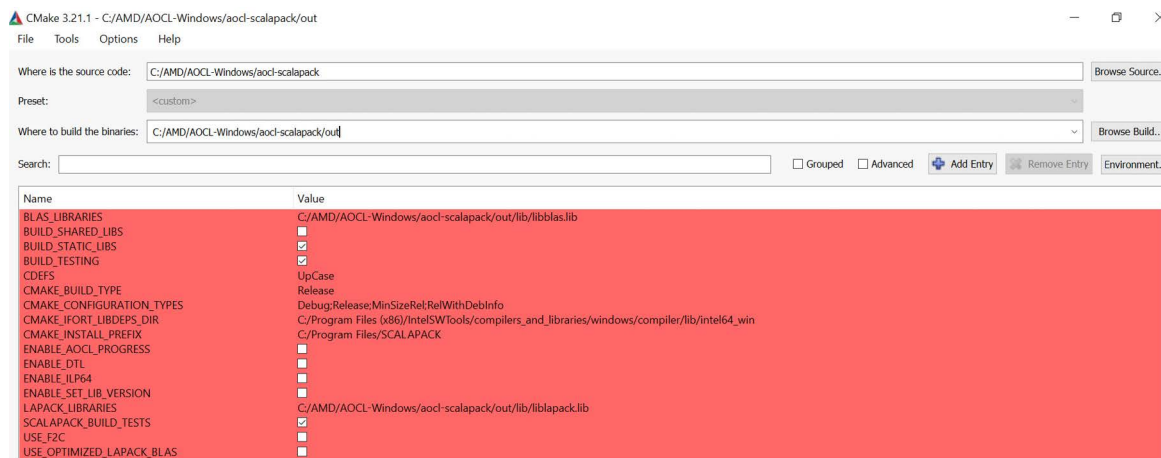
1. Set the **source** (folder containing aocl-scalapack source code) and **build** (folder in which the project files will be generated, for example, **out**) folder paths. It is not recommended to use the folder named **build** as a folder with that name exists at the top of AOCL-LAPACK source tree.
2. Click on the **Configure** button to prepare the project options.
3. Set the generator to **Visual Studio 17 2022** and the compiler to **ClangCl** or **LLVM**.

4. Update the options based on the project requirements. All the available options are listed in the following table:

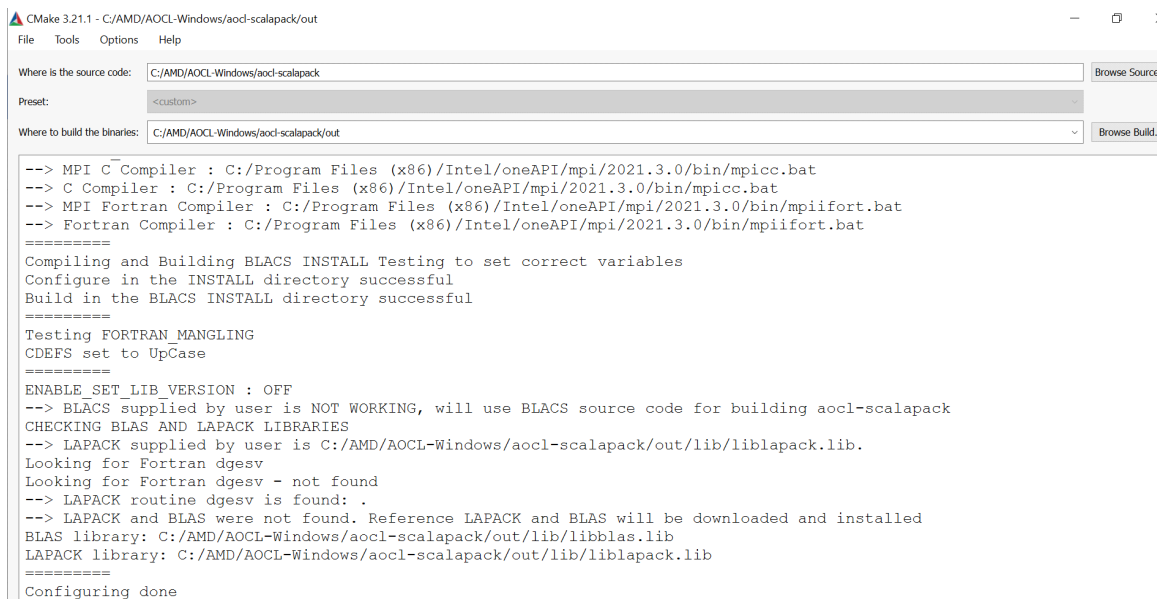
**Table 16. AOCL-ScaLAPACK CMake Parameter List**

Build Feature	CMake Command
Select debug or Release mode build	CMAKE_BUILD_TYPE=Debug/Release
Shared library	BUILD_SHARED_LIBS=ON BUILD_STATIC_LIBS=OFF
Static library	BUILD_STATIC_LIBS=ON BUILD_SHARED_LIBS=OFF
Provide external BLAS/AOCL-BLAS library	BLAS_LIBRARIES =<Path to BLAS/AOCL-BLAS lib>
Provide external LAPACK/AOCL-LAPACK library	LAPACK_LIBRARIES =<Path to lapack/AOCL-LAPACK lib>
Integer bit length: • ON => 64-bit integer length • OFF => 32-bit integer length	ENABLE_ILP64
Flags disabled by default	USE_OPTIMIZED_LAPACK_BLAS

## 5. Select the available and recommended options as follows:



**Figure 10. AOCL-ScaLAPACK CMake Options**



**Figure 11. AOCL-ScaLAPACK CMake Config Options**

## 6. Click the **Generate** button and then **Open Project**.

### 8.3.1.2 Building the Project in Visual Studio GUI

Complete the following steps in Microsoft Visual Studio GUI:

1. Open the project generated by CMake (build folder) in *“Preparing Project with CMake GUI” on page 94*.



2. To generate the AOCL-ScaLAPACK binaries, build the **ScaLAPACK** project. The library files would be generated in the folder **out** based on the project settings.

For example:

*aocl-scalapack/out/lib/Release/scalapack.lib*

*aocl-scalapack/out/Testing/Release/scalapack.dll*

### 8.3.2 Building AOCL-ScaLAPACK using Command-line Arguments

The project configuration and build procedures can be triggered from the command prompt as follows:

#### 8.3.2.1 Configuring the Project in Command Prompt

Complete the following steps to configure the project using the command prompt:

1. In the ScaLAPACK project folder, create a folder **out**.
2. Open the command prompt in that directory and run the following command:

```
cmake -S .. -B . -G "Visual Studio 17 2022" -DCMAKE_BUILD_TYPE=Release
-DBUILD_SHARED_LIBS=ON
-DBUILD_STATIC_LIBS=OFF -DBLAS_LIBRARIES="<path to AOCL-BLAS library>/AOCL-
LibBlis-Win-MT-dll.lib"
-DLAPACK_LIBRARIES="<path to AOCL-LAPACK library>/AOCL-LibFLAME-Win-MT-dll.lib"
```

Refer to [Table 16](#) to update the parameter options in the command according to the project requirements.

#### 8.3.2.2 Building the Project in Command Prompt

Complete the following steps to build the project using the command prompt:

1. Open command prompt in the *aocl-scalapack/out* directory.
2. Invoke CMake with the build command and release or debug option. For example:

```
cmake --build . --config Release
```

The library files would be generated inside the folder **Release** or **Debug** based on the project settings.

On Windows, the inbuilt communications submodule of ScaLAPACK, called Basic Linear Algebra Communication Subprograms (BLACS), exposes the API symbols in upper case without underscore format.

### 8.3.3 Building and Running the Individual Tests

Microsoft Visual Studio projects for the individual tests are generated as part of the CMake generate step. Refer the previous sections to build the test projects from Microsoft Visual Studio GUI or command prompt.

The test application binaries are generated in the folder `<aocl-scalapack>/out/Testing/Release` or `<aocl-scalapack>/out/Testing/Debug` based on the project settings. Run the tests from the command prompt as follows:

```
Release> mpiexec xcbrd.exe
```

## 8.4 Checking AOCL-ScaLAPACK Operation Progress

AOCL libraries perform tasks that can be computationally expensive. The AOCL Progress feature provides a mechanism, for a selected set of APIs, for the application to check how far a computation has progressed through a callback function.

### Usage

The application must define a callback function in a specific format and register this callback function with the AOCL-ScaLAPACK library.

The callback function prototype must be defined as follows:

```
int aocl_scalapack_progress(
const char* api,
const integer *lenapi,
const integer *progress,
const integer *mpi_rank,
const integer *total_mpi_processes
)
```

The following table explains AOCL-ScaLAPACK Progress feature callback function parameters:

**Table 17. AOCL-ScaLAPACK Progress Feature Callback Function Parameters**

Parameter	Purpose
api	Name of the API running currently
lenapi	Length of the API name character buffer
progress	Linear progress made in the current thread so far
mpi_rank	Current process rank
total_mpi_processes	Total number of processes used to perform the operation

### Callback Registration

The callback function must be registered with the library to report the progress:

```
aocl_scalapack_set_progress(aocl_scalapack_progress);
```

Example:

```
int AOCL_progress(const char* const api, const int *lenapi, const int *progress,
                  const int *mpi_rank, const int *total_mpi_processes)
{
    printf( "In AOCL Progress MPI Rank: %i API: %s progress: %i MPI processes: %i\n",
            *mpi_rank, api, *progress,*total_mpi_processes );
    return 0;
}
```

## Limitation

Currently, AOCL-ScaLAPACK progress feature is supported only on Linux.

## 8.5 Additional Features

The additional features supported at runtime through the environment variable setting are as follows:

**Table 18. Additional Features**

Feature	Description	Environment Variable	OS Support
Trace	Enable function call trace for double data type APIs.	AOCL_SL_TRACE	Linux, Windows
Log	Enable logging of input argument values of double data type APIs.	AOCL_SL_LOG	Linux, Windows
AOCL Progress	Check how far a computation has progressed through a callback function for 3 major factorization APIs (LU, QR, Cholesky ) for all data type variants	AOCL_SL_PROGRESS	Linux, Windows

Example:

- export AOCL\_SL\_LOG=1 in Linux enables the log file at run time.
- set AOCL\_SL\_PROGRESS=1 in Windows enables the AOCL Progress feature at run time.

---

## Chapter 9 AOCL-RNG

---

The AMD Random Number Generator (AOCL-RNG) library is a pseudorandom number generator library. It provides a comprehensive set of statistical distribution functions and various uniform distribution generators (base generators) including Wichmann-Hill, Mersenne Twister, and SIMD-based Fast Mersenne Twister (SFMT). The library contains six base generators and twenty-three distribution generators. In addition, you can supply a custom-built generator as the base generator for all the distribution generators.

### 9.1 Installation

**Note:** *AOCL-RNG can only be installed from pre-built binaries.*

The AOCL-RNG binary is available at the following URL:

<https://www.amd.com/en/developer/aocl/rng-library.html>

Also, AOCL-RNG binary can be installed from the AOCL master installer tar file available at the following URL:

<https://www.amd.com/en/developer/aocl.html>

The tar file includes pre-built binaries of other AMD libraries as explained in *"Using Master Package" on page 18*.

To install the AOCL-RNG binary for Windows, refer to *"Using Windows Packages" on page 22*. *rng\_amd.dll* and *rng\_amd.lib* are a part of the dynamic library and *rng\_amd-static.lib* is a static library.

As the AOCL-RNG library has a dependency on the AOCL-LibM and AOCL-BLAS libraries, note the following:

- To install AOCL-LibM binary for linux and Windows, refer to *"Installation" on page 85*.
- To install AOCL-BLAS binary for linux and Windows, refer to *"Using Pre-built Binaries" on page 26*.
- Those libraries must be linked with the application.

Set the runtime library search path (using the environment variable LD\_LIBRARY\_PATH) before running the application as follows:

```
$ export LD_LIBRARY_PATH=<path-to-aocl-libm-library-libamdlbm.so>:<path-to-aocl-blas-library-libblis.so>:$LD_LIBRARY_PATH
```

## 9.2 Using AOCL-RNG Library on Linux

To use the AOCL-RNG library in your application, link the library while building the application.

The following is a sample Makefile for an application that uses the AOCL-RNG library:

```
RNGDIR := <path-to-AOCL-RNG-library>
CC := gcc
CFLAGS := -I$(RNGDIR)/include
//CFLAGS For ILP64 case
//CFLAGS := -I$(RNGDIR)/include -DINTEGER64
CLINK := $(CC)
CLINKLIBS := -lamdlibm -lblis -lgfortran -lm -lrt -ldl
LIBRNG := $(RNGDIR)/lib/librng_amd.so
//Compile the program
$(CC) -c $(CFLAGS) test_rng.c -o test_rng.o
//Link the library
$(CLINK) test_rng.o $(LIBRNG) $(CLINKLIBS) -o test_rng.exe
```

For more information, refer the examples directory in the AOCL-RNG library install location.

## 9.3 Using AOCL-RNG Library on Windows

Complete the following steps to use AOCL-RNG library on Windows:

1. Create a 64-bit console app project in Visual Studio 17 2022.
2. Use the following navigation to select Clang-cl compiler:  
**Project > Properties > Configuration Properties > General > Platform Toolset > LLVM(Clang-cl)**
3. Use *example/\** sources as a reference to find the RNG API call flow.
4. Include the AOCL-RNG header file (*rng.h*) and call required AOCL-RNG APIs in Windows application.
5. Copy the AOCL-RNG header file (*rng.h*) and AOCL-RNG dll library (*rng\_amd.dll* and *rng\_amd.lib*) to the same project folder.
6. Copy AOCL-LibM DLL library (*libalm.dll* and *libalm.lib*) to the same project folder.
7. Copy AOCL-BLAS single-threaded DLL library (*AOCL-LibBlis-Win-dll.dll* and *AOCL-LibBlis-Win-dll.lib*) to the same project folder.
8. Use the following navigation to add WIN64 preprocessor definition:  
**Project > Properties > C/C++ > Preprocessor > Preprocessor Definitions**
9. Compile and then run the application.
10. You may create Fortran based project in similar manner and compile it using ifort compiler.
11. You can also compile your application using AOCL-RNG static library (*rng\_amd-static.lib*).

---

## Chapter 10 AOCL-SecureRNG

---

AOCL-SecureRNG is a library that provides the APIs to access the cryptographically secure random numbers generated by the AMD hardware based RNG. These are high quality robust random numbers designed for the cryptographic applications. The library makes use of RDRAND and RDSEED x86 instructions exposed by the AMD hardware. The applications can just link to the library and invoke a single or a stream of random numbers. The random numbers can be of 16-bit, 32-bit, 64-bit, or arbitrary size bytes.

### 10.1 Installation

The AOCL-SecureRNG library can be downloaded from following URL:

<https://www.amd.com/en/developer/aocl/rng-library.html>

Also, AMD SecureRNG can be installed from the AOCL master installer tar file available at the following URL:

<https://www.amd.com/en/developer/aocl.html>

The tar file includes pre-built binaries of other AMD libraries as explained in “*Using Master Package*” on page 18.

To install the AOCL-SecureRNG binary for Windows, refer to “*Using Windows Packages*” on page 22. *amdsecrng.dll* and *amdsecrng.lib* are a part of the dynamic library and *amdsec-static.lib* is a static library.

### 10.2 Usage

The following source files are included in the AOCL-SecureRNG package:

- *include/secrng.h* — A header file that has declaration of all the library APIs.
- *src\_lib/secrng.c* — Contains the implementation of the APIs.
- *src\_test/secrng\_test.c* — Test application to test all the library APIs.
- *Makefile* — To compile the library and test the application.

You can use the included *makefile* to compile the source files and generate dynamic and static libraries. Then, you can link it to your application and invoke the required APIs.

The following code snippet shows a sample usage of the library API:

```
//Check for RDRAND instruction support
int ret = is_RDRAND_supported();
int N = 1000;

//If RDRAND supported
if (ret == SECRNG_SUPPORTED)
{
    uint64_t rng64;

    //Get 64-bit random number
    ret = get_rdrand64u(&rng64, 0);

    if (ret == SECRNG_SUCCESS)
        printf("RDRAND rng 64-bit value %lu\n\n", rng64);
    else
        printf("Failure in retrieving random value using RDRAND!\n");

    //Get a range of 64-bit random values
    uint64_t* rng64_arr = (uint64_t*) malloc(sizeof(uint64_t) * N);

    ret = get_rdrand64u_arr(rng64_arr, N, 0);

    if (ret == SECRNG_SUCCESS)
    {
        printf("RDRAND for %u 64-bit random values succeeded!\n", N);
        printf("First 10 values in the range : \n");
        for (int i = 0; i < (N > 10? 10 : N); i++)
            printf("%lu\n", rng64_arr[i]);
    }
    else
        printf("Failure in retrieving array of random values using RDRAND!\n");
}
else
{
    printf("No support for RDRAND!\n");
}
```

In the example, `get_rdrand64u` is invoked to return a single 64-bit random value and `get_rdrand64u_arr` is used to return an array of 1000 64-bit random values.

## 10.3 Using AOCL-SecureRNG Library on Windows

Complete the following steps to use AOCL-SecureRNG library on Windows:

1. Create a 64-bit console app project in Visual Studio 17 2022.
2. Use the following navigation to select Clang-cl compiler:

**Project > Properties > Configuration Properties > General > Platform Toolset > LLVM (Clang-cl)**

3. Use `secrng_test.c` as a reference to find the AOCL-SecureRNG API call flow.

4. Include the AOCL-SecureRNG header file (*secrng.h*) and call required AOCL-SecureRNG APIs under window application.
5. Copy the AOCL-SecureRNG header file (*secrng.h*) and AOCL-SecureRNG DLL library (*amdsecrng.dll* and *amdsecrng.lib*) to same project folder.
6. Compile and then run the application.
7. You may create Fortran based project in similar manner and compile it using ifort compiler.
8. You can also compile your application using AOCL-SecureRNG static library (*amdsecrng-static.lib*).



## Chapter 11 AOCL-Sparse

---

AOCL-Sparse is a library containing basic linear algebra subroutines for sparse matrices and vectors. It is designed to be used with C and C++.

The current functionality of AOCL-Sparse is organized in the following categories:

- Sparse Level 3 functions describe the operations between a matrix in sparse format and a matrix in dense/sparse format.
- Sparse Level 2 functions describe the operations between a matrix in sparse format and a vector in dense format.
- Sparse Solver functions that perform matrix factorization and solution phases.
- Analysis and execute functionalities for performing optimized Sparse Matrix-Dense Vector multiplication and Sparse Solver.
- Sparse Format Conversion functions describe operations on a matrix in sparse format to obtain a different matrix format.

The list of supported functions is as follows:

- Sparse Level 3
  - `aoclsparse_xcsrmm` (single and double precision)
  - `aoclsparse_xcsr2m` (single and double precision)
- Sparse Level 2
  - `aoclsparse_xcsrmmv` (single and double precision)
  - `aoclsparse_xellmv` (single and double precision)
  - `aoclsparse_xdiamv` (single and double precision)
  - `aoclsparse_xbsrmv` (single and double precision)
  - `aoclsparse_xcsrsv` (single and double precision)
  - `aoclsparse_xtrsv` (single and double precision)
  - `aoclsparse_dmv` (double precision)
- Sparse Solvers
  - `aoclsparse_xilu_smoother`
  - `aoclsparse_xilu0`
  - `aoclsparse_itsol_d_rci_solve` (double precision)
  - `aoclsparse_itsol_s_rci_solve` (single precision)
  - `aoclsparse_itsol_d_solve` (double precision)
  - `aoclsparse_itsol_s_solve` (single precision)

- Sparse Auxiliary
  - `aoclsparse_get_version`
  - `aoclsparse_create_mat_descr`
  - `aoclsparse_destroy_mat_descr`
  - `aoclsparse_copy_mat_descr`
  - `aoclsparse_set_mat_fill_mode`
  - `aoclsparse_get_mat_fill_mode`
  - `aoclsparse_set_mat_diag_type`
  - `aoclsparse_get_mat_diag_type`
  - `aoclsparse_destroy_mat_csr`
  - `aoclsparse_destroy`
  - `aoclsparse_create_xcsr` (single and double precision)
- Conversion
  - `aoclsparse_csr2ell_width`
  - `aoclsparse_xcsr2ell` (single and double precision)
  - `aoclsparse_csr2dia_ndiag`
  - `aoclsparse_xcsr2dia` (single and double precision)
  - `aoclsparse_csr2bsr_nnz`
  - `aoclsparse_xcsr2bsr` (single and double precision)
  - `aoclsparse_xcsr2csc` (single and double precision)
  - `aoclsparse_xcsr2dense` (single and double precision)
- Analysis
  - `aoclsparse_set_mv_hint`
  - `aoclsparse_set_lu_smoother_hint`
  - `aoclsparse_set_mm_hint`
  - `aoclsparse_set_2m_hint`
  - `aoclsparse_optimize`

**Notes:**

1. *`aoclsparse_create_mat_csr` is not available from AOCL-Sparse 3.2 release. You can use the new function `aoclsparse_create_(s/d)csr` for creating a new matrix structure.*
2. *`aoclsparse_destroy_mat_csr` will be deprecated soon. You can use the new function `aoclsparse_destroy` for destroying the matrix structure and free internally allocated memory.*

## Multi-thread Support

AOCL-Sparse provides multi-thread support for specific APIs through OpenMP by default. You can set the total number of threads using the environment variables AOCLSPARSE\_NUM\_THREADS (recommended) or OMP\_NUM\_THREADS. If both environment variables are set, AOCL-Sparse library gives higher precedence to AOCLSPARSE\_NUM\_THREADS. If neither variable is set, the default number of threads is 1. The list of functions with multi-thread support are as follows:

- aoclsparse\_xcsrsv (single and double precision)
- aoclsparse\_xellmv (single and double precision)
- aoclsparse\_dmv (double precision)

For more information on performing multi-thread runs, refer *“Simple Test” on page 109*.

For more information on the AOCL-Sparse APIs, refer *AOCL-Sparse\_API\_Guide.pdf* in the docs folder (<https://github.com/amd/aocl-sparse>).

## 11.1 Installation

### 11.1.1 Building AOCL-Sparse from Source on Linux

The following prerequisites must be met:

- Git
- CMake versions 3.11 through 3.25
- Boost library versions 1.65 through 1.77

Complete the following steps to build different packages of the library, including dependencies and test application:

1. Install AOCL-BLAS and AOCL-LAPACK.
2. Define the environment variable AOCL\_ROOT to point to AOCL Libs installation:

```
export AOCL_ROOT=/opt/aocl
```

For the cases where AOCL\_ROOT cannot be exported by placing both AOCL-BLAS and AOCL-LAPACK libraries in the same path, you can use the following variables during the CMake configuration to point to the AOCL-BLAS or AOCL-LAPACK libraries and headers:

- AOCLSPARSE\_BLIS\_LIB
  - AOCLSPARSE\_FLAME\_LIB
  - AOCLSPARSE\_BLIS\_INCLUDE\_DIR
  - AOCLSPARSE\_FLAME\_INCLUDE\_DIR
3. Download the latest release of AOCL-Sparse (<https://github.com/amd/aocl-sparse>).
  4. Clone the Git repository (<https://github.com/amd/aocl-sparse.git>).

## 5. Run the command:

```
cd aocl-sparse
```

## 6. Create the build directory and change to it:

```
$ mkdir -p build/release
cd build/release
```

## 7. Run CMake as per the required compiler and library type:

**Table 19. Compiler and Library Type**

Compiler	Library Type	ILP 64 Support	Command
G++ (Default)	Static	OFF (Default)	cmake ../.. -DBUILD_SHARED_LIBS=OFF
		ON	cmake ../.. -DBUILD_SHARED_LIBS=OFF -DBUILD_ILP64=ON
	Shared (Default)	OFF (Default)	cmake ../..
		ON	\$ cmake ../.. -DBUILD_ILP64=ON
AOCC	Static	OFF (Default)	cmake ../.. -DCMAKE_CXX_COMPILER=clang++ - DBUILD_SHARED_LIBS=OFF
		ON	cmake ../.. -DCMAKE_CXX_COMPILER=clang++ - DBUILD_SHARED_LIBS=OFF -DBUILD_ILP64=ON
	Shared (Default)	OFF (Default)	cmake ../.. -DCMAKE_CXX_COMPILER=clang++
		ON	\$ cmake ../.. -DCMAKE_CXX_COMPILER=clang++ - DBUILD_ILP64=ON

## 8. Following CMake build options are applicable for Windows and Linux systems:

**Table 20. AOCL-Sparse - CMake Build Options**

Build Option	Feature
CMAKE_INSTALL_PREFIX	Use -DCMAKE_INSTALL_PREFIX=<path> to choose the custom path. The default install path is <i>/opt/aoclsparse/</i>
CMAKE_BUILD_TYPE	<ul style="list-style-type: none"> <li>Release =&gt; Release Library (Default)</li> <li>Debug =&gt; Debug Library</li> </ul>
CMAKE_CXX_COMPILER	Use -DCMAKE_CXX_COMPILER=clang++ for AOCC builds
BUILD_SHARED_LIBS	<ul style="list-style-type: none"> <li>OFF =&gt; Build Static Library</li> <li>ON =&gt; Build Dynamic/Shared library (Default)</li> </ul>

**Table 20. AOCL-Sparse - CMake Build Options**

Build Option	Feature
BUILD_ILP64	Integer length: <ul style="list-style-type: none"> <li>• OFF =&gt; 32-bit integer length (Default)</li> <li>• ON =&gt; 64-bit integer length</li> </ul>
SUPPORT_OMP	Multi-threading using OpenMP: <ul style="list-style-type: none"> <li>• OFF =&gt; Disable OpenMP</li> <li>• ON =&gt; Enable OpenMP (Default)</li> </ul>
USE_AVX512	<ul style="list-style-type: none"> <li>• OFF =&gt; Dynamically selects kernels (AVX2 and AVX512) for SpMV (Default)</li> <li>• ON =&gt; Enables AVX512 kernels for SpMV and TRSV</li> </ul>
BUILD_CLIENTS_BENCHMARKS	<ul style="list-style-type: none"> <li>• OFF =&gt; Disable building benchmarks (Default)</li> <li>• ON =&gt; Build client benchmarking (requires Boost library)</li> </ul>
BUILD_CLIENTS_SAMPLES	<ul style="list-style-type: none"> <li>• OFF =&gt; Disable building sparse API examples</li> <li>• ON =&gt; Enable building sparse examples for SPMV, CSR2M, DTRSV, CG, and GMRES (Default)</li> </ul>
BUILD_UNIT_TESTS	<ul style="list-style-type: none"> <li>• OFF =&gt; Unit tests are not built</li> <li>• ON =&gt; Unit testing is built and new target "test" is activated, this target should be used to test the correctness of the compiled library. It runs all the available executable targets and checks for success/failure of each test.</li> </ul>
BUILD_DOCS	<ul style="list-style-type: none"> <li>• ON =&gt; Build PDF and HTML documentation, this adds a new target "docs" (requires Linux and modern LaTeX distribution)</li> <li>• OFF =&gt; Does not activate the docs target (Default)</li> </ul>

9. Build the AOCL-Sparse library:

```
$ make -j$(nproc)
```

10. Install AOCL-Sparse to the directory `/opt/aoclsparse` or a custom path:

```
$ make install
```

## 11.1.2 Simple Test

After compiling the library with benchmarks, run the following AOCL-Sparse examples to test the installation:

1. Navigate to the test binary directory:

```
$ cd aocl-sparse/build/release/tests/staging
```

2. Ensure that the shared library is available in the library load path:

```
$ export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/path/to/libaoclsparse.so
```

3. Run CSR-SPMV on a randomly generated matrix to execute the aocl-sparse example:

```
$ ./aoclsparse-bench --function=csrcmv --precision=d --sizem=1000 --sizen=1000 --sizennz=4000 --verify=1
```

4. Run multi-threaded (4 threads) CSR-SPMV on a randomly generated matrix:

```
AOCLSPARSE_NUM_THREADS=4 numactl --physcpubind=4,5,6,7 ./aoclsparse-bench --function=csrcmv --precision=d --sizem=1000 --sizen=1000 --sizennz=4000 --verify=1
```

### 11.1.3 Using Pre-built Libraries

You can find the AMD optimized AOCL-Sparse source files at the following URL:

<https://github.com/amd/aocl-sparse/releases>

You can install the AOCL-Sparse binaries using the packages available at the following URL:

<https://www.amd.com/en/developer/aocl/sparse.html>

Also, you can install AOCL-Sparse binary from the AOCL master installer tar file available at the following URL:

<https://www.amd.com/en/developer/aocl.html>

The tar file includes pre-built binaries of other AMD libraries as explained in “Using Master Package” on page 18.

**Note:** For Windows pre-built binaries, only dynamic library is supported. Support for static library linking will be added in future releases.

## 11.2 Building AOCL-Sparse on Linux

You can find the sample programs demonstrating the usage of AOCL-Sparse APIs in the AOCL-Sparse source tests directory:

```
$ cd aocl-sparse/tests/examples
```

The sample programs are built as a part of AOCL-Sparse's CMake build system by enabling the flag BUILD\_CLIENTS\_SAMPLES and the binaries are located in <build\_directory>/tests/examples.

### 11.2.1 Use by Applications

To use AOCL-Sparse in your application, link the library while building the application. Configure the install directory using CMAKE\_INSTALL\_PREFIX and install the libraries using CMake build command:

```
cmake --build . --target install --config Release
```

Provide this install directory as the path to aocl sparse header and library.

Export the paths to AOCL-BLAS and AOCL-LAPACK library or headers in LD\_LIBRARY\_PATH and PATH variables:

```
export LD_LIBRARY_PATH=<path-to-aocl-blas-library>:<path-to-aocl-lapack-library>:$LD_LIBRARY_PATH

export PATH=<path-to-aocl-blas-headers>:<path-to-aocl-lapack-headers>:$PATH
```

## Examples to Build Sample Applications

SPMV:

```
g++ sample_spmv.cpp -I<path-to-aocl-sparse-header> -L<path-to-aocl-sparse-library>/-laoclsparse -o test_aoclsparse.x
```

CSR2M:

```
g++ sample_csr2m.cpp -I<path-to-aocl-sparse-header> -L<path-to-aocl-sparse-library>/-laoclsparse -o test_aoclsparse.x
```

TRSV:

```
g++ sample_dtrsv.cpp -I<path-to-aocl-sparse-header> -L<path-to-aocl-sparse-library>/-laoclsparse -o test_aoclsparse.x
```

CG example using Reverse Communication Interface(RCI):

```
g++ sample_itsol_d_cg_rci.cpp -I<path-to-aocl-sparse-header> -L<path-to-aocl-sparse-library>/-laoclsparse -o test_aoclsparse.x
```

CG example using Direct Interface:

```
g++ sample_itsol_d_cg.cpp -I<path-to-aocl-sparse-header> -L<path-to-aocl-sparse-library>/-laoclsparse -o test_aoclsparse.x
```

GMRES example using Reverse Communication Interface(RCI):

```
g++ sample_itsol_d_gmres_rci.cpp -I<path-to-aocl-sparse-header> -L<path-to-aocl-sparse-library>/-laoclsparse -o test_aoclsparse.x
```

GMRES example using Direct Interface:

```
g++ sample_itsol_d_gmres.cpp -I<path-to-aocl-sparse-header> -L<path-to-aocl-sparse-library>/-laoclsparse -o test_aoclsparse.x
```

The following is a sample *cpp* file depicting the AOCL-Sparse spmv API usage:

```
//file :sample_spmv.cpp
#include "aoclsparse.h"
#include <iostream>

#define M 5
#define N 5
#define NNZ 8

int main(int argc, char* argv[])
{
    aoclsparse_operation    trans    = aoclsparse_operation_none;

    double alpha = 1.0;
    double beta  = 0.0;

    // Print aoclsparse version
    std::cout << aoclsparse_get_version() << std::endl;

    // Create matrix descriptor
    aoclsparse_mat_descr descr;
    // aoclsparse_create_mat_descr set aoclsparse_matrix_type to aoclsparse_matrix_type_general
    // and aoclsparse_index_base to aoclsparse_index_base_zero.
    aoclsparse_create_mat_descr(&descr);

    aoclsparse_index_base base = aoclsparse_index_base_zero;

    // Initialise matrix
    // 1 0 0 2 0
    // 0 3 0 0 0
    // 0 0 4 0 0
    // 0 5 0 6 7
    // 0 0 0 0 8
    aoclsparse_int csr_row_ptr[M+1] = {0, 2, 3, 4, 7, 8};
    aoclsparse_int csr_col_ind[NNZ] = {0, 3, 1, 2, 1, 3, 4, 4};
    double         csr_val[NNZ] = {1, 2, 3, 4, 5, 6, 7, 8};
    aoclsparse_matrix A;
    aoclsparse_create_dcsr(A, base, M, N, NNZ, csr_row_ptr, csr_col_ind, csr_val);

    // Initialise vectors
    double x[N] = { 1.0, 2.0, 3.0, 4.0, 5.0};
    double y[M];

    //to identify hint id(which routine is to be executed, destroyed later)
    aoclsparse_set_mv_hint(A, trans, descr, 1);
```



```
// Optimize the matrix, "A"
aoclsparse_optimize(A);

std::cout << "Invoking aoclsparse_dmv..";
//Invoke SPMV API (double precision)
aoclsparse_dmv(trans,
&alpha,
A,
descr,
x,
&beta,
y);
std::cout << "Done." << std::endl;
std::cout << "Output Vector:" << std::endl;
for(aoclsparse_int i=0;i < M; i++)
std::cout << y[i] << std::endl;

aoclsparse_destroy_mat_descr(descr);
aoclsparse_destroy(A);
return 0;
}
```

A sample compilation command with the GCC compiler for the above code:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:<path-to-aocl-blas-library>:<path-to-aocl-lapack-
library>:<path-to-aocl-sparse-library>

g++ sample_csrmv.cpp -I<path-to-aocl-sparse-header> -L<path-to aocl-sparse-library> -
laoclsparse -o test_aoclsparse.x
```

## 11.3 Building AOCL-Sparse on Windows

GitHub URL: <https://github.com/amd/aocl-sparse>

AOCL-Sparse uses CMake along with Microsoft Visual Studio for building binaries from the sources on Windows. The following sections explain the GUI and command-line schemes of building the binaries and test suite.

### Prerequisites

For more information, refer to the Prerequisites sub-section in section *“Build AOCL-BLAS from Source on Windows” on page 49*.

## 11.3.1 Building AOCL-Sparse Using GUI

### 11.3.1.1 Preparing Project with CMake GUI

Complete the following steps to prepare the project with CMake GUI:

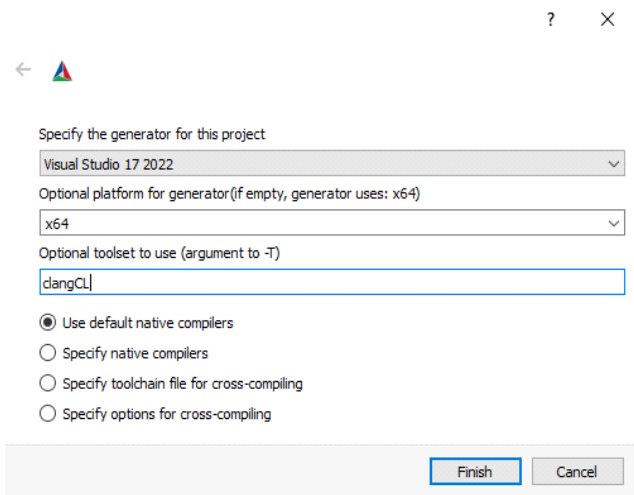
1. Install AOCL-BLAS and AOCL-LAPACK.
2. To link with dependent BLAS and LAPACK libraries, define the following CMake variables during configuration:

- AOCLSPARSE\_BLIS\_LIB
- AOCLSPARSE\_FLAME\_LIB
- AOCLSPARSE\_BLIS\_INCLUDE\_DIR
- AOCLSPARSE\_FLAME\_INCLUDE\_DIR

Launch CMake GUI using the Windows command line:

```
cmake-gui
```

3. Set the **source** (folder containing the AOCL-Sparse source code) and **build** (folder in which the project files will be generated) folder paths. It is not recommended to use the folder named **build** as it is already used for Linux build system.
4. Click on the **Configure** button to prepare the project options.
5. Set the generator to **Visual Studio 17 2022** and the platform toolset to **clangCL**:



**Figure 12. Specify Generator**

6. Update the options based on the project requirements. All the available options are listed in [Table 20](#). Select the available and recommended options as follows:

Name	Value
AMDCPU_TARGETS	
AOCLPARSE_CONFIG_DIR	\$(CMAKE_CURRENT_SOURCE_DIR)/lib
AOCL_BLI_INCLUDE_DIR	C:/Program Files/AMD/AOCL_Windows/amd-bli/include
AOCL_BLI_LIB	C:/Program Files/AMD/AOCL_Windows/amd-bli/lib/AOCL-LibBli-Win-MT.lib
AOCL_LIBFLAME	C:/Program Files/AMD/AOCL_Windows/amd-libflame/lib/AOCL-LibFlame-Win-MT.lib
AOCL_LIBFLAME_INCLUDE_DIR	C:/Program Files/AMD/AOCL_Windows/amd-libflame/include
ASAN	<input type="checkbox"/>
BUILD_CLIENTS_BENCHMARKS	<input checked="" type="checkbox"/>
BUILD_CLIENTS_SAMPLES	<input checked="" type="checkbox"/>
BUILD_DOCS	<input type="checkbox"/>
BUILD_LP64	<input type="checkbox"/>
BUILD_SHARED_LIBS	<input type="checkbox"/>
BUILD_SPARSE_EXAMPLE_OUTOFSRCTREE	<input type="checkbox"/>
BUILD_UNIT_TESTS	<input type="checkbox"/>
CMAKE_AOCL_ROOT	OFF
CMAKE_BUILD_TYPE	Release
CMAKE_CONFIGURATION_TYPES	Debug;Release;MinSizeRel;RelWithDebInfo
CMAKE_INSTALL_PREFIX	/package
COVERAGE	<input type="checkbox"/>
ENABLE_SET_BUILD_DATE	OFF
SUPPORT_OMP	<input checked="" type="checkbox"/>
USE_AVX512	<input type="checkbox"/>
VALGRIND	<input type="checkbox"/>

**Figure 13. AOCL-Sparse CMake Config Options**

**Note:** Currently, only single-threaded builds are supported. Multi-Threaded support will be added in future release.

7. Click the **Generate** button and then **Open Project**.

### 11.3.1.2 Building the Project in Visual Studio GUI

Complete the following steps in Microsoft Visual Studio GUI:

1. Open the AOCL-Sparse Visual Studio project from the build folder using the *aoclsparse.sln* file or the **Open Project** button in CMake GUI.
2. To generate the AOCL-Sparse binaries, choose the appropriate build configuration Debug or Release and then build the AOCL-Sparse project. The library files would be generated at `<build_dir>\library\Release`.

For example:

*aocl-sparse/build/library/Release/aoclsparse.dll*

*aoclsparse.lib*

### 11.3.2 Building AOCL-Sparse using Command-line Arguments

The project configuration and build procedures can be triggered from the command prompt as follows:

#### 11.3.2.1 Configuring the Project in Command Prompt

Complete the following steps to configure the project using command prompt:

1. Install AOCL-BLAS and AOCL-LAPACK.

2. To link with dependent BLAS and LAPACK libraries, define the following CMake variables during configuration:
  - AOCLSPARSE\_BLIS\_LIB
  - AOCLSPARSE\_FLAME\_LIB
  - AOCLSPARSE\_BLIS\_INCLUDE\_DIR
  - AOCLSPARSE\_FLAME\_INCLUDE\_DIR
3. In the AOCL-Sparse project folder, create a folder **out**.
4. Open the command prompt in the **out** directory and run the following command:

```
cmake .. -T ClangCL -G "Visual Studio 17 2022" -DCMAKE_CXX_COMPILER=ClangCL -
DCMAKE_INSTALL_PREFIX="<aocl_sparse_install_path>" -DSUPPORT_OMP=OFF -DAOCL_LIBFLAME="Lapack/
Library/with/path" -DAOCL_LIBFLAME_INCLUDE_DIR="path/to/Lapack/Headers" -DAOCL_BLIS_LIB="Blas/
Library/with/path" -DAOCL_BLIS_INCLUDE_DIR="path/to/Blas/Headers"
```

Refer to [Table 20](#) to update the parameter options in the command according to the project requirements.

**Note:** *Currently, only single-threaded builds are supported. Multi-Threaded support will be added in future release.*

### 11.3.2.2 Building the Project in Command Prompt

Complete the following steps to build the project using command prompt:

1. Open command prompt in the *aocl-sparse/out* directory.
2. Export the paths to the AOCL-BLAS and AOCL-LAPACK libraries:
 

```
set PATH=path\to\Lapack\Dynamic\library;path\to\BLAS\Dynamic\library;%PATH%
```
3. Export the paths to the AOCL-Sparse (from the build directory) that will be generated in step 5:
 

```
set PATH=<build_dir>/library/Release;%PATH%
```
4. If unit tests are enabled (BUILD\_UNIT\_TESTS=ON), then export the path to the GoogleTest libraries (from the build directory) that will be generated in step 5:
 

```
set PATH=<build_dir>/lib/Release;<build_dir>/bin/Release;%PATH%
```
5. Invoke CMake with the build command and release or debug option. For example:

```
cmake --build . --config Release
```

The library files would be generated inside the folder **Release** or **Debug** based on the project settings.

### 11.3.2.3 Building and Running the Test Suite

1. Microsoft Visual Studio projects for the individual tests are generated as a part of CMake generate step. Refer to the previous sections to build the test projects from Microsoft Visual Studio GUI or command prompt.

- Assuming that BUILD\_UNIT\_TESTS was enabled during CMake configuration/build steps, AOCL-Sparse unit tests (gtest and ctests) can be run for LP/STATIC and ILP/STATIC configurations in a single threaded configuration:

```
cd <build_directory>
ctest -VV
```

**Note:** Other configurations, such as Multi-threading and DYNAMIC builds are not yet supported.

## 11.4 Running an Individual AOCL-Sparse Test

The AOCL-Sparse executable accepts 2 types of inputs, namely randomly generated matrix data and matrices in Matrix Market format (mtx). The MTX inputs can be downloaded from SuiteSparse Matrix Collection website (<https://sparse.tamu.edu/>). Usage of both the type of inputs is shown below.

### 11.4.1 Run the Test on Linux

Complete the following steps to run the test on Linux:

- Export the paths to AOCL-BLAS and AOCL-LAPACK libraries and headers in the LD\_LIBRARY\_PATH and PATH variables:

```
export LD_LIBRARY_PATH=<path-to-aocl-blas-library>:<path-to-aocl-lapack-library>:<path-to-aocl-SPARSE-library>:$LD_LIBRARY_PATH
export PATH=<path-to-aocl-BLAS-headers>:<path-to-aocl-LAPACK-headers>:<path-to-aocl-SPARSE-headers>:$PATH
```

- Navigate to the AOCL-Sparse executable and run the test from the command prompt as follows:

**Random Data:**

```
./aoclsparse-bench.exe --function=optmv --precision=d --size=1000 --size=1000 --sizennz=4000 --verify=1
```

**MTX Input:**

```
./aoclsparse-bench.exe --function=optmv --precision=d --mtx=LFAT5.mtx --verify=1
```

- Run the multi-threaded (4 threads) CSR-SPMV on a randomly generated matrix:

```
export AOCLSPARSE_NUM_THREADS=4
./aoclsparse-bench.exe --function=csrvm --precision=d --size=1000 --size=1000 --sizennz=4000 --verify=1
```

## 11.4.2 Run the test on Windows

Complete the following steps to run the test on Windows:

1. Export the paths to AOCL-BLAS and AOCL-LAPACK libraries and headers in the PATH variable:

```
set PATH=<path-to-aocl-blas-library>;<path-to-aocl-lapack-library>;<path-to-aocl-SPARSE-library>;<path-to-aocl-blas-headers>;<path-to-aocl-lapack-headers>;<path-to-aocl-SPARSE-headers>;%PATH%
```

2. Navigate to the AOCL-Sparse executable and run the test from the command prompt as follows:

**Random Data:**

```
.\aoclsparse-bench.exe --function=optmv --precision=d --sizem=1000 --sizen=1000 --sizennz=4000 --verify=1
```

**MTX Input:**

```
./aoclsparse-bench.exe --function=optmv --precision=d --mtx=LFAT5.mtx --verify=1
```

3. Run the multi-threaded (4 threads) CSR-SPMV on a randomly generated matrix:

```
set AOCLSPARSE_NUM_THREADS=4
.\aoclsparse-bench.exe --function=csrvm --precision=d --sizem=1000 --sizen=1000 --sizennz=4000 --verify=1
```

## Chapter 12 AOCL-LibMem

---

AOCL-LibMem is a Linux library of data movement and manipulation functions (such as `memcpy()` and `strcpy()`) highly optimized for AMD Zen micro-architecture. This library has multiple implementations of each function that can be chosen based on the application requirements as per alignments, instruction choice, threshold values, and tunable parameters. It supports AVX2 and AVX512 CPU features. By default, it will choose the best fit implementation based on the underlying micro-architectural support for CPU features and instructions.

This release of the AOCL-LibMem library supports the following functions:

- `memcpy`
- `mempcpy`
- `memmove`
- `memset`
- `memcmp`
- `strcpy`

### 12.1 Building AOCL-LibMem for Linux

Minimum software requirements for compilation:

- GCC 12.2
- AOCC 4.0
- Python 3.6
- CMake 3.10

Complete the following steps to build AOCL-LibMem for Linux:

1. Download and install the AOCL master installer (*aocl-linux-`<compiler>`-`<version>`.tar.gz*) from:  
<https://www.amd.com/en/developer/aocl.html>
2. Locate the *aocl-libmem* folder in the root directory.
3. Create build directory:

```
$ mkdir build  
$ cd build
```

#### 4. Configure for one of the following builds as required:

##### Shared Library Configuration:

###### – GCC

```
# Default Native Build
$ cmake -D CMAKE_C_COMPILER=gcc ../aocl-libmem

# Cross Compiling AVX2 Binary on AVX512 Machine
$ cmake -D CMAKE_C_COMPILER=gcc -D ALMEM_ARCH=avx2 ../aocl-libmem

# Cross Compiling AVX512 Binary on AVX2 Machine
$ cmake -D CMAKE_C_COMPILER=gcc -D ALMEM_ARCH=avx512 ../aocl-libmem

# Enabling Tunable Parameters
$ cmake -D CMAKE_C_COMPILER=gcc -D ENABLE_TUNABLES=Y ../aocl-libmem
```

###### – AOCC (Clang)

```
# Default Native Build
$ cmake -D CMAKE_C_COMPILER=clang ../aocl-libmem

# Cross Compiling AVX2 Binary on AVX512 Machine
$ cmake -D CMAKE_C_COMPILER=clang -D ALMEM_ARCH=avx2 ../aocl-libmem

# Cross Compiling AVX512 Binary on AVX2 Machine
$ cmake -D CMAKE_C_COMPILER=clang -D ALMEM_ARCH=avx512 ../aocl-libmem

# Enabling Tunable Parameters
$ cmake -D CMAKE_C_COMPILER=clang -D ENABLE_TUNABLES=Y ../aocl-libmem
```

##### Static Library Configuration:

###### – GCC

```
# Default Native Build
$ cmake -D CMAKE_C_COMPILER=gcc -D BUILD_SHARED_LIBS=N ../aocl-libmem

# Cross Compiling AVX2 Binary on AVX512 Machine
$ cmake -D CMAKE_C_COMPILER=gcc -D ALMEM_ARCH=avx2 -D BUILD_SHARED_LIBS=N ../aocl-libmem

# Cross Compiling AVX512 Binary on AVX2 Machine
$ cmake -D CMAKE_C_COMPILER=gcc -D ALMEM_ARCH=avx512 -D BUILD_SHARED_LIBS=N ../aocl-libmem

# Enabling Tunable Parameters
$ cmake -D CMAKE_C_COMPILER=gcc -D ENABLE_TUNABLES=Y -D BUILD_SHARED_LIBS=N ../aocl-libmem
```



### – AOCC (Clang)

```
# Default Native Build
$ cmake -D CMAKE_C_COMPILER=clang -D BUILD_SHARED_LIBS=N ../aocl-libmem

# Cross Compiling AVX2 Binary on AVX512 Machine
$ cmake -D CMAKE_C_COMPILER=clang -D ALMEM_ARCH=avx2 -D BUILD_SHARED_LIBS=N ../aocl-libmem

# Cross Compiling AVX512 Binary on AVX2 Machine
$ cmake -D CMAKE_C_COMPILER=clang -D ALMEM_ARCH=avx512 -D BUILD_SHARED_LIBS=N ../aocl-libmem

# Enabling Tunable Parameters
$ cmake -D CMAKE_C_COMPILER=clang -D ENABLE_TUNABLES=Y -D BUILD_SHARED_LIBS=N ../aocl-libmem
```

### 5. Build:

```
$ cmake --build .
```

### 6. Install:

```
$ make install
```

After compilation:

- The shared library file *libaocl-libmem.so* will be saved in <build/lib/shared>.
- The static library file *libaocl-libmem.a* will be saved in <build/lib/static>.

**Note:** *Dynamic Dispatcher is not supported. Hence, it is recommended not to load/run the AVX512 library on a non-AVX512 machine as it will lead to crash due to unsupported instructions.*

## 12.2 Running an Application

The applications can preload the AOCL-LibMem shared library to replace the standard c library memory functions for better performance gains on AMD “Zen” micro-architectures.

To run the application, preload the *libaocl-libmem.so* generated from the build procedure above:

```
$ LD_PRELOAD=<path to build/lib/shared/libaocl-libmem.so> <executable> <params>
```

## 12.3 Running an Application with Tunables

LibMem built with tunables enabled exposes two tunable parameters that will help you select the implementation of your choice:

- **LIBMEM\_OPERATION:** Instruction based on alignment and cacheability
- **LIBMEM\_THRESHOLD:** The threshold for ERMS and Non-Temporal instructions

Following two states are possible with this library based on the tunable settings:

- **Default State:** None of the parameters is tuned.
- **Tuned State:** One of the parameters is tuned with a valid option.

### 12.3.1 Default State

In this state, none of the parameters are tuned; the library will pick up the best implementation based on the underlying AMD “Zen” micro-architecture.

Run the application by preloading the tunables enabled *libaocl-libmem.so*:

```
$ LD_PRELOAD=<path to build/lib/shared/libaocl-libmem.so> <executable> <params>
```

### 12.3.2 Tuned State

In this state, one of the parameters is tuned by the application at run time. The library will choose the implementation based on the valid tuned parameter at run time. Only one of the tunable can be set to a valid set of format/options as described in Table 21.

#### 12.3.2.1 LIBMEM\_OPERATION

You can set the tunable LIBMEM\_OPERATION as follows:

```
LIBMEM_OPERATION=<operations>,<source_alignment>,<destination_alignmnet>
```

Based on this option, the library chooses the best implementation based on the combination of move instructions, alignment of the source and destination addresses.

#### Valid Options

- <operations> = [avx2|avx512|erms]
- <source\_alignment> = [b|w|d|q|x|y|n]
- <destination\_alignmnet> = [b|w|d|q|x|y|n]

Use the following table to select the right implementation for your application:

**Table 21. Application Implementations**

Application Requirement	LIBMEM_OPERATION	Instructions	Side-effects
Vector unaligned source and destination	[avx2 avx512],b,b	Load:VMOVDQU; Store:VMOVDQU	None
Vector aligned source and destination	[avx2 avx512],y,y	Load:VMOVDQA; Store:VMOVDQA	Unaligned source and/or destination address will lead to crash
Vector aligned source and unaligned destination	[avx2 avx512],y,[b w d q x]	Load:VMOVDQA; Store:VMOVDQU	None
Vector unaligned source and aligned destination	[avx2 avx512],[b w d q x],y	Load:VMOVDQU; Store:VMOVDQA	None

**Table 21. Application Implementations**

Application Requirement	LIBMEM_OPERATION	Instructions	Side-effects
Vector non temporal load and store	[avx2 avx512],n,n	Load:VMOVNTDQ A; Store:VMOVNTDQ	Unaligned source and/or destination address will lead to crash
Vector non temporal load	[avx2 avx512],n,[b w d q x y]	Load:VMOVNTDQ A; Store:VMOVDQU	None
Vector non temporal store	[avx2 avx512],[b w d q x y],n	Load:VMOVDQU; Store:VMOVNTDQ	None
Rep movs unaligned source or destination	erms,b,b	REP MOVSB	None
Rep movs word aligned source and destination	erms,w,w	REP MOVSW	Data corruption or crash if the length is not a multiple of 2
Rep movs double word aligned source and destination	erms,d,d	REP MOVSD	Data corruption or crash if the length is not a multiple of 4
Rep movs quad word aligned source and destination	erms,q,q	REP MOVSQ	Data corruption or crash if the length is not a multiple of 8

**Note:** A best-fit solution for the underlying micro-architecture will be chosen if the tunable is in an invalid format.

For example, to use only avx2-based move operations with both unaligned source and aligned destination addresses:

```
$ LD_PRELOAD=<build/lib/shared/libaocl-libmem.so> LIBMEM_OPERATION=avx2,b,y <executable>
```

### 12.3.2.2 LIBMEM\_THRESHOLD

You can set the tunable LIBMEM\_THRESHOLD as follows:

```
LIBMEM_THRESHOLD=<repmov_start_threshold>,<repmov_stop_threshold>,<nt_start_threshold>,<nt_stop_threshold>
```

Based on this option, the library will choose the implementation with tuned threshold settings for supported instruction sets: {vector, rep mov, non-temporal}.

#### Valid Options

- <repmov\_start\_threshold> = [0, +ve integers]
- <repmov\_stop\_threshold> = [0, +ve integers, -1]
- <nt\_start\_threshold> = [0, +ve integers]

- `<nt_stop_threshold> = [0, +ve integers, -1]`

Where, -1 refers to the maximum length.

Refer the following table for the sample threshold settings:

**Table 22. Sample Threshold Settings**

LIBMEM_THRESHOLD	Vector Range	RepMov Range	Non-Temporal Range
0,2048,1048576,-1	(2049, 1048576)	[0,2048]	[1048576, max value of unsigned long long)
0,0,1048576,-1	[0,1048576)	[0,0]	[1048576, max value of unsigned long long)

**Note:** A system configured threshold will be chosen if the tunable is in an invalid format.

For example, to use **\*\*REP MOVE\*\*** instructions for a range of 1KB to 2KB and `non_temporal` instructions for a range of 512 KB and above:

```
$ LD_PRELOAD=<build/lib/shared/libaocl-libmem.so> LIBMEM_THRESHOLD=1024,2048,524288,-1
<executable>
```

## Chapter 13 AOCL-Cryptography

---

AOCL-Cryptography is a library consisting of the core cryptographic functions optimized for AMD “Zen” micro-architecture. This library has multiple implementations of different:

- Advanced Encryption Standard (AES) encryption/decryption ciphers
- Secure Hash Algorithms (SHA-2 and SHA-3)
- Cipher and Hash based Message Authentication Code (MAC) algorithms
- Elliptic-curve Diffie–Hellman (ECDH) and Rivest, Shamir, and Adleman (RSA) key generation functions

The AOCL-Cryptography library has the following functions:

- AES encrypt/decrypt routines for the following cipher schemes:
  - Cipher Block Chaining (CBC)
  - Cipher Feedback (CFB)
  - Output Feedback (OFB)
  - Counter (CTR)
  - Galois/Counter Mode (GCM)
  - Ciphertext Stealing Mode (XTS)
  - Counter with Cipher Block Chaining Message Authentication Code (CCM)
- SHA-2 digest routines for the following schemes:
  - SHA2\_224
  - SHA2\_256
  - SHA2\_384
  - SHA2\_512
- SHA-3 digest routines for the following schemes:
  - SHA3\_224, SHA3\_256, SHA3\_384, and SHA3\_512
  - SHAKE128 and SHAKE256
- Hash-based Message Authentication Code (HMAC) routines for the following schemes:
  - HMAC\_SHA2\_224, HMAC\_SHA2\_256, HMAC\_SHA2\_384, and HMAC\_SHA2\_512
  - HMAC\_SHA3\_224, HMAC\_SHA3\_256, HMAC\_SHA3\_384, and HMAC\_SHA3\_512
- Cipher-based Message Authentication Code (CMAC) routines for the following schemes:
  - CMAC\_AES\_128
  - CMAC\_AES\_192
  - CMAC\_AES\_256

- AES - SIV (Synthetic IV)
- ECDH x25519 key exchange functions:
  - Generate Public Key
  - Compute Secret Key
- RSA
  - Encrypt with public Key
  - Decrypt with private Key

**Note:** Only non-padded mode is supported in AOCL 4.1 release.

## 13.1 Requirements

- CMake 3.14
- GCC 11.1.0
- OpenSSL v3.0.0 through 3.0.7
- Clang 15 on Windows
- [AOCL-Utills library](#)
- For more information on supported Linux operating systems, refer to [Operating Systems library](#).

## 13.2 Installation

### 13.2.1 Building AOCL-Cryptography from Source on Linux

Complete the following steps to build AOCL-Cryptography from source on Linux:

1. GitHub URL: <https://github.com/amd/aocl-crypto>
2. Clone the repository aocl-crypto.
3. `cd aocl-crypto`
4. `mkdir build`
5. `cd build`
6. Run the configure command `cmake ../` using the following options:

**Table 23. AOCL-Cryptography - Linux Options**

Option	Description
ALCP_ENABLE_EXAMPLES (ON/OFF)	Compile the example code
CMAKE_BUILD_TYPE (Debug/Release)	Specify the build type
ENABLE_AOCL_CPUID (ON/OFF)	Enable CPUID functions from the AOCL-Utills library
AOCL_CPUID_INSTALL_DIR	AOCL-Utills installation path

**Table 23. AOCL-Cryptography - Linux Options**

Option	Description
OPENSSL_INSTALL_DIR	OpenSSL (3.0.0 or later) installation path
CMAKE_INSTALL_PREFIX	AOCL-Cryptography installation path
ALCP_SANITIZE (ON/OFF)	Enable sanitizers (asan, tsan, and so on)
AOCL_COMPAT_LIBS	Supported values= ipp,openssl/ipp/openssl  Enable compilation of IPP OpenSSL provider libraries. <i>Notes:</i> <ol style="list-style-type: none"> <li>1. This CMake option is supported only with the GCC compiler in 4.1 release.</li> <li>2. The IPP header files should be added to the CPLUS_INCLUDE_PATH environment variable (working version for IPP is 2021_7).</li> <li>3. OpenSSL provider support is not yet enabled for ECDH, RSA, and SIV functions.</li> <li>4. IPP provider support is not yet enabled for ECDH, RSA, SIV, and CCM functions.</li> </ol>
ALCP_ENABLE_DOXYGEN	Values: ON/OFF  Enable Doxygen documentation generation. <i>Note: Doxygen version supported: v1.9.6 or later.</i>

7. `make -j$(nproc)`

8. `make install`

### Testing Examples

1. Navigate to the installed directory.
2. Ensure that AOCL and OpenSSL lib directories are added to LD\_LIBRARY\_PATH and LIBRARY\_PATH environment variables:

```
export LD_LIBRARY_PATH=<path to aocl crypto lib>:<path to OpenSSL lib>:$LD_LIBRARY_PATH;
export LIBRARY_PATH=<path to aocl crypto lib>:<path to OpenSSL lib>:$LIBRARY_PATH;
```

3. `make`

4. Run the executables generated in `./bin/<module>`. For example, `./bin/mac/hmac`.

## 13.2.2 Building AOCL-Cryptography from Source on Windows

AOCL-Cryptography requires CMake and Microsoft Visual Studio for building the binaries from the sources on Windows.

### Prerequisites

- CMake versions 3.0 through 3.26.1

- Microsoft Visual Studio 2019 (build 16.8.7) through 2022 (build 17.5.3)
- Desktop development with C++: C++ Clang-tools for windows (x64 or x86)
- LLVM plug-in for Microsoft Visual Studio (if the latest version of LLVM is installed separately, this plug-in enables linking Microsoft Visual Studio with the installed LLVM toolchain)
- Install OpenSSL (3.0.0 or later) and add *openssl\bin* path to the PATH environment variables, if not set

## Configure and Build

1. Clone the repository aocl-crypto.
2. Open Command Prompt or PowerShell.
3. `cd aocl-crypto`
4. `mkdir build`
5. Run `cmake` configure using the following options:

**Table 24. AOCL-Cryptography - Windows Options**

Option	Description
CMAKE_BUILD_TYPE (Debug/Release)	Specify the build type
ALCP_ENABLE_EXAMPLES (ON/OFF)	Compile the example code
ENABLE_AOCL_CPUID (ON/OFF)	Enable CPUID functions from the AOCL-Utills library
AOCL_CPUID_INSTALL_DIR	AOCL-Utills installation path
OPENSSL_INSTALL_DIR	OpenSSL (3.0.0 or later) installation path
-A (platform)	x86/x64
-B (build directory)	Build
-T (toolset)	ClangCl/LLVM
-G (specify generator)	Visual Studio 17 2022/ Visual Studio 16 2019

6. build the library:

```
--config=release/debug
PS>cmake --build ./build --config=release -j
```

## Testing Examples

1. Navigate to the build directory.
2. Ensure that the *lib/Release* directory is added to PATH environment variables.
3. If not set already, add *openssl\bin* path to the PATH environment variables.
4. Run the executables generated in *.\examples\<module>\Release\\*.exe*.

Example: *.\examples\cipher\Release\aes-ccm.exe*



## 13.3 Using AOCL-Cryptography in a Sample Application

A few pointers for using AOCL-Cryptography in a sample application:

- For using the encrypt/decrypt routines, use the header file in the test application:

*include/alcp/alcp.h*

An example to use the cipher routines can be found in:

*aocl-crypto/examples/cipher*

- For using the digest routines, use the header file:

*include/alcp/digest.h*

An example to use the digest routines can be found in:

*aocl-crypto/examples/digest*

### 13.3.1 Compiling and Running Examples

Complete the following steps to compile and run the AOCL-Cryptography examples from the downloaded packages:

1. Download and untar the aocl-crypto package.
2. `cd amd-crypto`
3. `make`
4. To run example applications (for digest):

```
LD_LIBRARY_PATH=<path to aocl crypto lib>:<path to openssl lib> ./bin/digest/sha2_384_example
```

### 13.3.2 AOCL-Cryptography Library Provider for OpenSSL

For more information on usage instructions, refer to the following URL:

<https://github.com/amd/aocl-crypto/blob/main/docs/compat/openssl.pdf>

### 13.3.3 Integrating AOCL Libraries with Applications that Use IPP

For more information, refer to the following URL:

<https://github.com/amd/aocl-crypto/blob/main/docs/compat/ipp.pdf>

## Chapter 14 AOCL-Compression

---

AOCL-Compression is a software framework of various lossless data compression and decompression methods tuned and optimized for AMD “Zen”-based CPUs. This library suite supports the following:

- Linux and Windows platforms.
- lz4, zlib/deflate, lzma, zstd, bzip2, snappy, and lz4hc optimized compression and decompression methods.
- A unified standardized API set and the existing native APIs of the respective methods.
- Dynamic dispatcher feature that executes the most optimal function variant implemented using Function Multi-versioning and hence, offering a single optimized library portable across different x86 CPU architectures.
- A test suite is provided for validation and performance benchmarking of the supported compression and decompression methods. The test suite also supports the benchmarking of IPP compression methods, such as lz4, lz4hc, bzip2, and zlib on the Linux-based platforms.
- The library build framework offers CTest based testing of the test cases that are implemented using GTest and the library test suite.
- A Python-based performance benchmarking automation script is provided for benchmarking needs.
- Doxygen based documentation covering library's API level details.
- Custom build options supported to exclude the unnecessary compression methods from the library build for achieving a lower code footprint.

### 14.1 Installation

#### 14.1.1 Using Pre-built Libraries

The library and test bench binary for Linux and Windows can be installed from one of the following:

- AOCL-Compression page (<https://developer.amd.com/amd-aocl/aocl-compression/>)
- AOCL master installer: tar and zip packages for Linux and Windows respectively (<https://developer.amd.com/amd-aocl/>)

### 14.1.2 Building from Source

Complete the following steps to build AOCL-Compression from source:

1. Download the AOCL-Compression source package from GitHub (<https://github.com/amd/aocl-compression>).
2. Follow the steps in the *README* file to build the library for Linux or Windows.

While building with AOCC 4.1, disable the CMake configuration option

`ENABLE_STRICT_WARNINGS` to avoid reporting of warnings related to `-Wstrict-prototypes` `-Wno-deprecated-non-prototype` as errors.

It is recommended to use the unified APIs of the library, but the native APIs of the respective compression methods can be directly used by the applications. The compiled and installed package of the library that is generated has the known issue of missing out a few header files. You must copy the following header files to the compiled package folder *aocl\_compression/include* that is created at the installed path based on the prefix option:

- *algos/lzma/7zTypes.h*
- *algos/snappy/snappy-stubs-public.h*
- *algos/zlib/zconf.h*

## 14.2 Running AOCL-Compression Test Bench on Linux

Test bench supports several options to validate, benchmark, or debug the supported compression methods. It uses the unified API set to invoke the compression methods supported by AOCL-Compression. It can also invoke and benchmark some of the IPP's compression methods.

To check the various options supported by the test bench, use one of the following commands:

```
aocl_compression_bench -h
Or
aocl_compression_bench --help
```

Use the following command for an example to run the test bench and validate the outputs from all the supported compression and decompression methods for a given input file:

```
aocl_compression_bench -a -t <input filename>
```

Use the following command for an example to run the test bench and check the performance of a particular compression and decompression method for a given input file:

```
aocl_compression_bench -ezstd:5:0 -p <input filename>
```

Here, 5 is the level and 0 is the additional parameter to specify the custom window size for the ZSTD method.

To run the test bench with *error/debug/trace/info logs*, use the command:

```
aocl_compression_bench -a -t -v <input filename>
```

Here, you can pass `-v` with a number such as `v<n>` that can take the following values:

- 1 for Error (default)
- 2 for Info
- 3 for Debug
- 4 for Trace

To test and benchmark the performance of IPP's compression methods, use the test bench option `-c` along with the other relevant options (as explained above).

Currently, IPP's lz4, lz4hc, bzip2, and zlib methods are supported by the test bench.

Complete the following steps:

1. Set the library path environment variable (export `LD_LIBRARY_PATH` on Linux) to point to the installed IPP library path.

Alternatively, you can also run `vars.sh` that comes along with the IPP installation to setup the environment variable.

2. Download lz4-1.9.3, zlib-1.2.11, and bzip2-1.0.8 source packages.
3. Apply IPP's patch files as follows:

```
patch -p1 <"path to corresponding patch file">
```

4. Build the patched IPP lz4, bzip2, and zlib libraries as per the steps in the IPP README files (in the corresponding patch file locations) for these compression methods.
5. Set the library path environment variable (export `LD_LIBRARY_PATH` on Linux) to point to the patched IPP lz4, bzip2, and zlib libraries.
6. Run the test bench to benchmark IPP library methods as follows:

```
aocl_compression_bench -a -p -c <input filename>
aocl_compression_bench -elz4 -p -c <input filename>
aocl_compression_bench -elz4hc -p -c <input filename>
aocl_compression_bench -ezlib -p -c <input filename>
aocl_compression_bench -ebzip2 -p -c <input filename>
```

For more information, refer to the README file available with the source package in GitHub (<https://github.com/amd/aocl-compression>).

## 14.3 Running AOCL-Compression Test Bench on Windows

Test bench on Windows supports all the user options as on Linux, except for the `-c` option to link and test the IPP's compression methods. For more information, refer to [“Running AOCL-Compression Test Bench on Linux” on page 131](#).

To set and launch the test bench with a specific user option:

1. Go to project `aocl_compression_bench` > **Properties** > **Debugging**.
2. Specify the user options and the input test file.

For more information, refer to the README file available with the source package at GitHub (<https://github.com/amd/aocl-compression>).

## 14.4 API Reference

### 14.4.1 Unified Standardized API Set

```
//Interface API to compress data
int64_t aocl_llc_compress(aocl_compression_desc *handle,
                        aocl_compression_type codec_type);

//Interface API to decompress data
int64_t aocl_llc_decompress(aocl_compression_desc *handle,
                          aocl_compression_type codec_type);

//Interface API to setup the compression method
void aocl_llc_setup(aocl_compression_desc *handle,
                  aocl_compression_type codec_type);

//Interface API to destroy the compression method
void aocl_llc_destroy(aocl_compression_desc *handle,
                    aocl_compression_type codec_type);

//Interface API to get compression library version string
const char *aocl_llc_version(void);
```

### 14.4.2 Interface Data Structures

```
//Types of compression methods supported
typedef enum
{
    LZ4 = 0,
    LZ4HC,
    LZMA,
    BZIP2,
    SNAPPY,
    ZLIB,
    ZSTD,
    AOCL_COMPRESSOR_ALGOS_NUM
} aocl_compression_type;
```

```

typedef struct
{
    char *inBuf;           /**< Pointer to input buffer data          */
    char *outBuf;          /**< Pointer to output buffer data         */
    char *workBuf;         /**< Pointer to temporary work buffer      */
    size_t inSize;         /**< Input data length                    */
    size_t outSize;        /**< Output data length                   */
    size_t level;          /**< Requested compression level          */
    size_t optVar;          /**< Additional variables or parameters   */
    int numThreads;         /**< Number of threads available for multi-threading */
    int numMPIranks;        /**< Number of available multi-core MPI ranks */
    size_t memLimit;        /**< Maximum memory limit for compression/decompression */
    int measureStats;       /**< Measure speed and size of compression/decompression */
    uint64_t cSize;         /**< Size of compressed output            */
    uint64_t dSize;         /**< Size of decompressed output          */
    uint64_t cTime;         /**< Time to compress input               */
    uint64_t dTime;         /**< Time to decompress input             */
    float cSpeed;           /**< Speed of compression                 */
    float dSpeed;           /**< Speed of decompression                */
    int optOff;             /**< Turn off all optimizations           */
    int optLevel;           /**< Optimization level: \n
                                0 - non-SIMD algorithmic optimizations, \n
                                1 - SSE2 optimizations, \n
                                2 - AVX optimizations, \n
                                3 - AVX2 optimizations, \n
                                4 - AVX512 optimizations
                                */
    int printDebugLogs;     /**< Print debug logs                     */
} aocl_compression_desc;

```

### 14.4.3 Library Return Error Codes

```
typedef enum
{
    ERR_UNSUPPORTED_METHOD = -3,      ///

```

### 14.4.4 Native APIs

```
//bzip2 Interface API to compress data
int BZ2_bzBuffToBuffCompress(
char*      dest,
unsigned int* destLen,
char*      source,
unsigned int sourceLen,
int        blockSize100k,
int        verbosity,
int        workFactor
);

//bzip2 Interface API to decompress data
int BZ2_bzBuffToBuffDecompress (
char*      dest,
unsigned int* destLen,
char*      source,
unsigned int sourceLen,
int        small,
int        verbosity
);
```

```
//lz4 Interface API to compress data
int LZ4_compress_default(
    const char* src,
    char* dst,
    int srcSize,
    int dstCapacity
);

//lz4 Interface API to decompress data
int LZ4_decompress_safe (
    const char* src,
    char* dst,
    int compressedSize,
    int dstCapacity
);

//lz4hc Interface API to compress data
int LZ4_compress_HC(
    const char* src,
    char* dst,
    int srcSize,
    int dstCapacity,
    int compressionLevel
);

//lz4hc Interface API to decompress data
int LZ4_decompress_safe (
    const char* src,
    char* dst,
    int compressedSize,
    int dstCapacity
);

//lzma Interface API to compress data
int LzmaEncode(
    Byte *dest, SizeT *destLen, const Byte *src, SizeT srcLen,
    const CLzmaEncProps *props, Byte *propsEncoded, SizeT *propsSize, int writeEndMark,
    ICompressProgress *progress, ISzAllocPtr alloc, ISzAllocPtr allocBig
);

//lzma Interface API to decompress data
int LzmaDecode(
    Byte *dest, SizeT *destLen, const Byte *src, SizeT *srcLen,
    const Byte *propData, unsigned propSize, ELzmaFinishMode finishMode,
    ELzmaStatus *status, ISzAllocPtr alloc
);
```



```
//snappy Interface API to compress data
void RawCompress(
const char* input,
size_t input_length,
char* compressed,
size_t* compressed_length
);

//snappy Interface API to decompress data
bool RawUncompress(
const char* compressed, size_t compressed_length,
char* uncompressed
);

//zlib Interface API to compress data
int compress2(
unsigned char *dest, unsigned long *destLen,
const unsigned char *source, unsigned long sourceLen,
int level
);

//zlib Interface API to decompress data
int uncompress(
unsigned char *dest, unsigned long *destLen,
const unsigned char *source, unsigned long sourceLen
);

//zstd Interface API to compress data
size_t ZSTD_compress_advanced(
ZSTD_CCtx* cctx,
void* dst, size_t dstCapacity,
const void* src, size_t srcSize,
const void* dict, size_t dictSize,
ZSTD_parameters params
);

//zstd Interface API to decompress data
size_t ZSTD_decompressDctx(
ZSTD_DCtx* dctx,
void* dst, size_t dstCapacity,
const void* src, size_t srcSize
);
```

## 14.4.5 Example Test Program

The following test program shows the sample usage and calling sequence of aocl-compression APIs to compress and decompress a test input:

```
#include <stdio.h>
#include "aocl_compression.h"

int main (int argc, char **argv)
{
    aocl_compression_desc aocl_compression_ds;
    aocl_compression_desc *aocl_compression_handle = &aocl_compression_ds;
    FILE *inFp = NULL;
    int file_size = 0;
    char *inPtr = NULL, *compPtr = NULL, *decompPtr = NULL;
    int64_t resultComp = 0, resultDecomp = 0;

    if (argc < 2)
    {
        printf("Provide input test file path\n");
        return -1;
    }
    inFp = fopen(argv[1], "rb");
    fseek(inFp, 0L, SEEK_END);
    file_size = ftell(inFp);
    rewind(inFp);

    // One of the compression methods as per aocl_compression_type
    aocl_compression_type method = LZ4;
    aocl_compression_handle->level = 0;
    aocl_compression_handle->optVar = 0;
    aocl_compression_handle->printDebugLogs = 0;
    aocl_compression_handle->inSize = file_size;
    aocl_compression_handle->outSize = (file_size + (file_size / 6) + (16 * 1024));
    inPtr = (char *)calloc(1, aocl_compression_handle->inSize);
    compPtr = (char *)calloc(1, aocl_compression_handle->outSize);
    decompPtr = (char *)calloc(1, aocl_compression_handle->inSize);
    aocl_compression_handle->inBuf = inPtr;
    aocl_compression_handle->outBuf = compPtr;
    file_size = fread(inPtr, 1, file_size, inFp);

    // 1. setup and create a handle
    aocl_llc_setup(aocl_compression_handle, method);

    // 2. compress
    resultComp = aocl_llc_compress(aocl_compression_handle, method);

    if (resultComp <= 0)
    {
        printf("Compression: failed\n");
        goto error_exit;
    }
    printf("Compression: done\n");
}
```

```
// decompress
aocl_compression_handle->inSize = resultComp;
aocl_compression_handle->outSize = file_size;
aocl_compression_handle->inBuf = compPtr;
aocl_compression_handle->outBuf = decompPtr;

resultDecomp = aocl_llc_decompress(aocl_compression_handle, method);

if (resultDecomp <= 0)
{
    printf("Decompression Failure\n");
    goto error_exit;
}
printf("Decompression: done\n");

// destroy handle
aocl_llc_destroy(aocl_compression_handle, method);
error_exit:
if (inPtr)
    free(inPtr);
if (compPtr)
    free(compPtr);
if (decompPtr)
    free(decompPtr);
return 0;
}
```

To build this example test program on a Linux system using GCC or AOCC, you must specify the *aocl\_compression.h* header file and link the *libaocl\_compression.so* file as follows:

```
gcc test.c -I<aocl_compression.h file path> -L <libaocl_compression.so file path> -
laocl_compression
```

## 14.5 Optional Optimization Options

Some additional optimization options are supported in the library that can give performance benefits based on specific test conditions. These optional features are not enabled by default and must be turned on depending on their need:

**Table 25. Optional Optimization Options**

Option	Description
AOCL_LZ4_OPT_PREFETCH_BACKWARDS	Enable LZ4 optimizations related to backward prefetching of data (Disabled by default)
SNAPPY_MATCH_SKIP_OPT	Enable Snappy match skipping optimization (Disabled by default)
LZ4_FRAME_FORMAT_SUPPORT	Enable building LZ4 with Frame format and API support (Enabled by default)
AOCL_LZ4HC_DISABLE_PATTERN_ANALYSIS	Disable Pattern Analysis in LZ4HC for level 9 (Enabled by default)

**Table 25. Optional Optimization Options**

Option	Description
AOCL_ZSTD_4BYTE_LAZY2_MATCH_FINDER	Enable 4-byte comparison for finding a potential better match candidate with Lazy2 compressor (Disabled by default)
AOCL_TEST_COVERAGE	Enable GTest and AOCL test bench based CTest suite (Disabled by default)
BUILD_DOC	Build documentation for this library (Disabled by default)
ZLIB_DEFLATE_FAST_MODE_2	Enable optimization for deflate fast using Z_FIXED strategy. Do not combine with ZLIB_DEFLATE_FAST_MODE_3 (Disabled by default)
ZLIB_DEFLATE_FAST_MODE_3	Enable ZLIB deflate quick strategy. Do not combine with ZLIB_DEFLATE_FAST_MODE_2 (Disabled by default)
AOCL_LZ4_MATCH_SKIP_OPT_LDS_STRAT1	Enable LZ4 match skipping optimization strategy-1 based on a larger base step size applied for long distance search (Disabled by default)
AOCL_LZ4_MATCH_SKIP_OPT_LDS_STRAT2	Enable LZ4 match skipping optimization strategy-2 by aggressively setting search distance on top of strategy-1. Recommended to be used with Silesia corpus. (Disabled by default)
AOCL_EXCLUDE_BZIP2	Exclude BZIP2 compression method from the library build (Disabled by default)
AOCL_EXCLUDE_LZ4	Exclude LZ4 compression method from the library build. LZ4HC also gets excluded (Disabled by default)
AOCL_EXCLUDE_LZ4HC	Exclude LZ4HC compression method from the library build (Disabled by default)
AOCL_EXCLUDE_LZMA	Exclude LZMA compression method from the library build (Disabled by default)
AOCL_EXCLUDE_SNAPPY	Exclude SNAPPY compression method from the library build (Disabled by default)
AOCL_EXCLUDE_ZLIB	Exclude ZLIB compression method from the library build (Disabled by default)
AOCL_EXCLUDE_ZSTD	Exclude ZSTD compression method from the library build (Disabled by default)

## Chapter 15 AOCL-Utills

---

AOCL-Utills provides a uniform interface to all the AOCL libraries to access the CPU features for AMD CPUs. This library provides the following features:

- Core details
- Flags available/usable
- ISA available/usable
- Topology about L1/L2/L3 caches

AOCL-Utills is designed for integration with the other AOCL libraries. Each project has its own mechanism to identify CPU and provide necessary features such as Dynamic Dispatch. The main purpose of this library is to provide a centralized mechanism to update/validate and provide information to the users.

This is the first release of the AOCL-Utills library and it supports the following functions:

- ISA available/usable
- API to check following features:
  - SHA, AES, and VAES availability
  - RDSEED and RDRAND availability
  - AVX2 availability
  - AVX512 foundation and sub-feature flags
- APIs for cache topology

**Note:** *This library detects only the CPUs of AMD "Zen" architecture, there are no plans to add support for other x86 implementations of other CPU vendors. Some of the utilities may fail or behave in an unexpected manner on the predecessors of AMD "Zen" architecture.*

### 15.1 Requirements

- CMAKE v3.15 or later
- GCC v12.2 or later
- Clang v15 or later
- Refer to [“Build Utilities” on page 17](#) for Make and Microsoft Visual Studio versions
- For more information on the supported operating systems, refer to [“Operating Systems” on page 16](#)
- stdc++ library must be linked when using the AOCL-Utills static binary

## 15.2 Clone and Build the AOCL-Utills Library

Complete the following steps to clone and build the AOCL-Utills library:

1. Download the latest release of AOCL-Utills (<https://github.com/amd/aocl-utils>).
2. Clone the Git repository (<https://github.com/amd/aocl-utils.git>).
3. Run the command:

```
cd aocl-utils
```

4. For more information on the detailed steps to build and install AOCL-Utills (based on OS, compilers, and so on) refer to the *aocl-utils/BUILD.md* file.

**Note:** For installing the AOCL-Utills library with Spack on Linux-based environment, refer to “Building from Source” on page 18.

## 15.3 Using AOCL-Utills

For this library, C++ is used for implementation. This library also provides C interfaces for the calls from other C programs/libraries. After installing the AOCL-Utills library:

- For using the C++ routines, use the *include/alci.h* header file that has classes/members to get CPU features, AMD "Zen" micro-architecture and cache information.
- For using the C routines:
  - Use *include/alci/arch.h* to get the CPU features and AMD "Zen" micro-architecture information.
  - Use *include/alci/cache.h* to get the Cache topology (L1, L2, and L3) information.

### 15.3.1 C API Example

Example: *test\_c\_application.c*

```
#include "alci/arch.h"
#include "alci/alci.h"
#include <stdio.h>

int main(int argc, char **argv) {

    if (alcpu_is_amd()) {
        printf("Is CPU based on AMD Zen: true\n");
    } else {
        printf("Is CPU based on AMD Zen: false\n");
    }

    return 0;
}
```

### 15.3.2 C++ API Example

Example: *test\_cpp\_application.cc*

```
#include "alci/cxx/alci.hh"
#include "alci/cxx/cpu.hh"
#include <stdio.h>

alci::Cpu CpuData = alci::Cpu();

int main(int argc, char **argv) {

    if (CpuData.isAmd()) {
        printf("Is CPU based on AMD Zen: true\n");
    } else {
        printf("Is CPU based on AMD Zen: false\n");
    }

    return 0;
}
```

### 15.3.3 Building on Windows

On Windows, you can build an application with the AOCL-Utills library using Clang/Clang++ Compilers as follows:

1. Create a 64-bit console app C++ project in Microsoft Visual Studio 17 2022.
2. To select Clang-cl compiler, navigate to **Project > Properties > Configuration Properties > General > Platform Toolset > LLVM(Clang-cl)** or **llvm**.
3. Use *test\_c\_application.c* or *test\_cpp\_application.cc* sources as a reference for the API call flow of AOCL-Utills.
4. Add them into project using:  
**Project > Add Existing item** > select *test\_c\_application.c* or *test\_cpp\_application.cc* from the project source directory.
5. Include the AOCL-Utills header files (*such as include/alci/alci.h, include/alci/cxx/alci.hh, and so on*) and call the required AOCL-Utills APIs in the Windows application.
6. Update the include path in:  
**Project > Properties > C/C++ > General > Additional Include Directories**
7. Update the AOCL-Utills library path (where *libaoclutils.lib* or *libaoclutils\_static.lib* exist) in:  
**Project > Properties > Linker > General > Additional Library Directories**
8. Update the AOCL-Utills library name in:  
**Project > Properties > Linker > Input > Additional Dependencies** (*libaoclutils.lib* or *libaoclutils\_static.lib*)

9. If AOCL-Utills dynamic library is used, copy the AOCL-Utills DLL library (*libaoclutils.dll*) to the same project application folder.
10. Compile the project and run the application.

### 15.3.4 Building on Linux

On Linux, you can build an application with the AOCL-Utills library using:

- GCC/G++ Compilers:

```
# Export the libaoclutils binaries path into LD_LIBRARY_PATH variable.
export LD_LIBRARY_PATH=<path of libaoclutils binaries>:${LD_LIBRARY_PATH}
```

#### Using Static Library:

```
gcc -std=gnu11 test_c_application.c -o test_c_application.exe -L<path of libaoclutils binaries>
-l:libaoclutils.a -lstdc++ -I<path of libaoclutils include directory>
```

```
g++ -std=gnu++17 test_cpp_application.cc -o test_cpp_application.exe -L<path of libaoclutils
binaries> -l:libaoclutils.a -I<path of libaoclutils include directory>
```

#### Using Dynamic/Shared Library:

```
gcc -std=gnu11 test_c_application.c -o test_c_application.exe -L<path of libaoclutils binaries>
-l:libaoclutils.so -I<path of libaoclutils include directory>
```

```
g++ -std=gnu++17 test_cpp_application.cc -o test_cpp_application.exe -L<path of libaoclutils
binaries> -l:libaoclutils.so -I<path of libaoclutils include directory>
```

- AOCC Clang/Clang++ Compilers:

#### Using Static Library:

```
clang -std=c11 test_c_application.c -o test_c_application.exe -L<path of libaoclutils binaries>
-l:libaoclutils.a -lstdc++ -I<path of libaoclutils include directory>
```

```
clang++ -std=c++17 test_cpp_application.cc -o test_cpp_application.exe -L<path of libaoclutils
binaries> -l:libaoclutils.a -I<path of libaoclutils include directory>
```

#### Using Dynamic/Shared Library:

```
clang -std=c11 test_c_application.c -o test_c_application.exe -L<path of libaoclutils binaries>
-l:libaoclutils.so -I<path of libaoclutils include directory>
```

```
clang++ -std=c++17 test_cpp_application.cc -o test_cpp_application.exe -L<path of libaoclutils
binaries> -l:libaoclutils.so -I<path of libaoclutils include directory>
```

Similarly, to use AOCL-Utills in other libraries, link the AOCL-Utills binary using `-L` or `-l` flag and include the header files using `-I` flag.



### 15.3.5 Output

Finally, run *test\_c\_application.exe* or *test\_cpp\_application.exe* on system and that'll give the following output:

```
Is CPU based on AMD Zen: true for AMD Zen based CPU.  
Is CPU based on AMD Zen: false for other non-AMD Zen based CPU.
```

## Chapter 16 Linking AOCL to Applications

---

This section provides examples of how AOCL can be linked with the HPL benchmark and MUMPS sparse solver library.

### 16.1 High-performance LINPACK Benchmark (HPL)

HPL is a software package that solves a (random) dense linear system in double precision (64-bits) arithmetic on distributed memory computers. It is a LINPACK benchmark that measures the floating-point rate of execution for solving a linear system of equations.

To build an HPL binary from the source code, edit the MPxxx and LAXxx directories in your architecture-specific Makefile to match the installed locations of your MPI and Linear Algebra library. For AOCL-BLAS, use the F77 interface with `F2CDEFS = -DAdd__ -DF77_INTEGER=int -DStringSunStyle`.

Use the multi-threaded AOCL-BLAS with the following configuration for an optimal performance:

```
./configure --enable-cblas -t openmp --disable-sup-handling --prefix=<path> auto
```

Setup HPL.dat before running the benchmark.

#### 16.1.1 Configuring HPL.dat

*HPL.dat* file contains the configuration parameters. The important parameters are Problem Size, Process Grid, and BlockSize.

- Problem Size (N) — For best results, the problem size must be set large enough to use 80-90% of the available memory.
- Process Grid (P and Q) — P x Q must match the number of MPI ranks. P and Q must be as close to each other as possible. If the numbers cannot be equal, Q must be larger.
- BlockSize (NB) — HPL uses the block size for the data distribution and for the computational granularity. Set NB=240 for an optimal performance.
- Set BCASTs=2 — Increasing-2-ring (2rg) broadcast algorithm gives a better performance than the default broadcast algorithm.

## 16.1.2 Running the Benchmark

The combination of multi-threading (through OpenMP library) and MPI is important to configure for optimal performance. Set the number of MPI tasks to number of L3 caches in the system for optimal performance.

The HPL benchmark typically produces a better single node performance number with the following configurations depending on which generation of AMD EPYC™ processor is used:

- 2<sup>nd</sup> Gen AMD EPYC™ Processors (codenamed “Rome”)

A dual socket AMD EPYC 7742 system consists of 32 CCXs, each having an L3 cache and a total of 2 x 64 cores (four cores per CCX). For maximum performance, use 32 MPI ranks with 4 OpenMP threads. Each MPI rank is bonded to 1 CCX and 4 threads per L3 cache.

Set the following flags while building and running the tests:

```
export BLIS_IC_NT=4
export BLIS_JC_NT=1
```

Execute the following command to run the test:

```
mpirun -np 32 --report-bindings --map-by ppr:1:l3cache,pe=4 -x OMP_NUM_THREADS=4 -x
OMP_PROC_BIND=TRUE -x OMP_PLACES=cores ./xhpl
```

BLIS\_IC\_NT and BLIS\_JC\_NT parameters are set for DGEMM parallelization at each shared L3 cache to improve the performance further.

- 3<sup>rd</sup> Gen AMD EPYC™ Processors (codenamed “Milan”)

The number of MPI ranks and maximum thread count per MPI rank depends on the specific EPYC SKU. For better performance, bind each MPI rank to a CCX, if there are 4 OpenMP threads. However, if 8 threads are used, then you should specify CCD instead of CCX.

Set the following flags while building and running the tests:

```
export BLIS_IC_NT=8
export BLIS_JC_NT=1
```

Execute the following command to run the test:

```
mpirun -np 16 --report-bindings --map-by ppr:1:l3cache,pe=8 -x OMP_NUM_THREADS=8 -x
OMP_PROC_BIND=TRUE -x OMP_PLACES=cores ./xhpl
```

## 16.2 MUMPS Sparse Solver Library

MUltifrontal Massively Parallel Solver (MUMPS: <http://mumps-solver.org/>) is an open-source package for solving systems of linear equations of the form:

$$Ax = b$$

Where, A is a square sparse matrix that can be one of the following on distributed memory computers:

- Unsymmetric
- Symmetric positive definite
- General symmetric

MUMPS implements a direct method based on a multi-frontal approach which performs the Gaussian factorization:

$$A = LU$$

Where, L is a lower triangular matrix and U an upper triangular matrix.

If the matrix is symmetric then the factorization:

$$A = LDLT$$

Where, D is a block diagonal matrix performed.

The system  $Ax = b$  is solved in the following steps:

### 1. Analysis

During an analysis, preprocessing including re-ordering and a symbolic factorization are performed. This depends on the external libs METIS, SCOTCH, and PORD (inside MUMPS source).  $A_{pre}$  denotes the preprocessed matrix.

### 2. Factorization

During the factorization,  $A_{pre} = LU$  or  $A_{pre} = LDLT$ , depending on the symmetry of the preprocessed matrix, is computed. The original matrix is first distributed (or redistributed) onto the processors depending on the mapping computed during the analysis. The numerical factorization is then a sequence of dense factorization on the frontal matrices.

### 3. Solution

The solution xpre of:

$LUxpre = bpre$  or  $LDLT xpre = bpre$

Where, xpre and bpre are the transformed solution x and right-hand side b respectively. They are associated to the preprocessed matrix Apre and obtained through the forward elimination step:

$Ly = bpre$  or  $LDy = bpre$

Followed by the backward elimination step:

$Uxpre = y$  or  $L^T xpre = y$ .

The solution xpre is finally processed to obtain the solution x of the original system  $Ax = b$ .

The AOCL libraries can be integrated with the MUMPS sparse solver to perform highly optimized linear algebra operations on AMD “Zen”-based processors.

## 16.2.1 Enabling AOCL with MUMPS

### 16.2.1.1 Using Spack On Linux

Complete the following steps to enable AOCL with MUMPS on Linux:

1. Set up Spack on the target machine.
2. Link the AOCL libraries AOCL-BLAS, AOCL-LAPACK, and AOCL-ScaLAPACK while installing MUMPS. Use the following Spack commands to install MUMPS with:
  - gcc compiler:
 

```
$ spack install mumps ^amdblis ^amdlibflame ^amdscalapack
```
  - aocc compiler:
 

```
$ spack install mumps ^amdblis ^amdlibflame ^amdscalapack %aocc
```
  - To use an external reordering library (for example, METIS), run the following command:
 

```
$ spack install mumps ^metis ^amdblis ^amdlibflame ^amdscalapack
```

### 16.2.1.2 On Windows

GitHub URL: <https://github.com/amd/mumps-build>

#### Prerequisites

Ensure that the following prerequisites are met:

- CMake and Ninja Makefile Generator — Ensure that Ninja is installed/updated in the Microsoft Visual Studio installation folder:

*C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\Common7\IDE\CommonExtensions\Microsoft\CMake\Ninja*

- Download the latest Binary Ninja from the URL:  
<https://github.com/ninja-build/ninja/releases>
- Intel® oneAPI toolkit must include C, C++, Fortran Compilers, and MPI. For more information, refer Intel documentation (<https://www.intel.com/content/www/us/en/developer/articles/tool/oneapi-standalone-components.html>).
- Pre-built AOCL libraries for AOCL-BLAS, AOCL-LAPACK, and AOCL-ScaLAPACK.
- If reordering library is METIS, complete the following steps:
  - a. Download the pre-built METIS library from SuiteSparse public repository (<https://github.com/egrup-gu/SuiteSparse.git>).
  - b. Build METIS library from the *metis* folder:  

```
cd SuiteSparse\metis-5.1.0
```
  - c. Define **IDXTYPEWIDTH** and **REALTYPEWIDTH** to 32 or 64 based on the required integer size in *metis/include/metis.h*.
  - d. Configure:  

```
cmake S . -B ninja_build_dir -G "Ninja" -DBUILD_SHARED_LIBS=OFF  
-DCMAKE_BUILD_TYPE=Release -DCMAKE_VERBOSE_MAKEFILE:BOOL=ON
```
  - e. Build the project:  

```
cmake --build ninja_build_dir --verbose
```

The library *metis.lib* is generated in *ninja\_build\_dir\lib*.
- Boost libraries on Windows:
  - Required to read the *.mtx* files efficiently and quickly
  - Essential for the test application *aocl\_amd.cpp* that links to MUMPS libraries and measures the performance for an Symmetric Positive Definite (SPD) *.mtx* file
  - Download the latest sources and bootstrap respectively from:  
<https://www.boost.org/users/download/>  
[https://www.boost.org/doc/libs/1\\_81\\_0/more/getting\\_started/windows.html#simplified-build-from-source](https://www.boost.org/doc/libs/1_81_0/more/getting_started/windows.html#simplified-build-from-source)
  - Define **BOOST\_ROOT** in *tests/CMakeLists.txt*

## Building MUMPS Sources

Complete the following steps to build the MUMPS sources on Windows:

1. Checkout the MUMPS build repository from AOCL GitHub (<https://github.com/amd/mumps-build>).
2. Open Intel oneAPI command prompt for Intel 64 for Microsoft Visual Studio 2019 from Windows search box.
3. Edit the default options in *options.cmake* in *mumps/cmake/*.
4. Remove any build directory if it exists already.

## 5. Configure the MUMPS project using Ninja:

```
cmake S . -B ninja_build_dir -G "Ninja" -DENABLE_AOCL=ON -DENABLE_MKL=OFF -DBUILD_TESTING=ON
-DCMAKE_INSTALL_PREFIX="/mumps/install/path" -Dscotch=ON -Dopenmp=ON -DBUILD_SHARED_LIBS=OFF
-Dparallel=ON -DCMAKE_VERBOSE_MAKEFILE:BOOL=ON -DCMAKE_BUILD_TYPE=Release
-DUSER_PROVIDED_BLIS_LIBRARY_PATH="<path/to/AOCL-BLAS/library/path>"
-DUSER_PROVIDED_BLIS_INCLUDE_PATH="<path/to/AOCL-BLAS/headers/path>"
-DUSER_PROVIDED_LAPACK_LIBRARY_PATH="<path/to/aocl-lapack/library/path>"
-DUSER_PROVIDED_LAPACK_INCLUDE_PATH="<path/to/aocl-lapack/headers/path>"
-DUSER_PROVIDED_SCALAPACK_LIBRARY_PATH="<path/to/scalapack/library/path>"
-DUSER_PROVIDED_METIS_LIBRARY_PATH="<path/to/metis/library/path>"
-DUSER_PROVIDED_METIS_INCLUDE_PATH="<path/to/metis/include/path>"
-DCMAKE_C_COMPILER="icx.exe" -DCMAKE_CXX_COMPILER="icx.exe"
-DCMAKE_Fortran_COMPILER="ifx.exe" -DBoost_ROOT="<path/to/boost_1_77_0>" -Dintsize64=OFF -
DUSER_PROVIDED_IMPI_LIB_ILP64_PATH="<path/to/boost>"
-DMUMPS_UPSTREAM_VERSION="5.5.1"
```

The following options are enabled in the command:

- **-DENABLE\_AOCL=ON:** <Enable AOCL Libraries>
- **-DENABLE\_MKL=OFF:** <Enable MKL Libraries>
- **-DBUILD\_TESTING=ON:** <Enable Mumps linking to test application to test>
- **-Dscotch=ON:** <Enable Metis Library for Reordering>
- **-Dopenmp=ON:** <Enable Multithreading using openmp>
- **-Dintsize64=OFF:** <Enable LP64 i.e., 32-bit integer size>
- **-DBUILD\_SHARED\_LIBS=OFF:** <Enable Static Library>
- **-Dparallel=ON:** <Enable Multithreading>
- **-DCMAKE\_VERBOSE\_MAKEFILE:BOOL=ON:** <Enable verbose build log>
- **-DCMAKE\_BUILD\_TYPE= Release:** <Enable Release build>
- **-DUSER\_PROVIDED\_BLIS\_LIBRARY\_PATH=** “<path/to/blas/lib>”
- **-DUSER\_PROVIDED\_BLIS\_INCLUDE\_PATH=** “<path/to/blas/header>”
- **-DUSER\_PROVIDED\_LAPACK\_LIBRARY\_PATH=** “<path/to/lapack/lib >”
- **-DUSER\_PROVIDED\_LAPACK\_INCLUDE\_PATH=** “<path/to/lapack/include/header
- **-DUSER\_PROVIDED\_SCALAPACK\_LIBRARY\_PATH=** “<path/to/scalapack/lib
- **-DUSER\_PROVIDED\_METIS\_LIBRARY=** “<Metis/library/with/absolute/path >”
- **-DUSER\_PROVIDED\_METIS\_LIBRARY\_PATH=** “<path/to/metis/lib>”
- **-DUSER\_PROVIDED\_METIS\_INCLUDE\_PATH=** “<path/to/metis/header>”
- **-DCMAKE\_C\_COMPILER=** “<intel c compiler>”
- **-DCMAKE\_Fortran\_COMPILER=** “<intel fortran compiler>”
- **-DBoost\_ROOT=** “<path/to/BOOST/INSTALLATION>”
- **-DUSER\_PROVIDED\_IMPI\_LIB\_ILP64\_PATH=**“<path/to/64-bit/Intel IMPI Library>”
- **-DMUMPS\_UPSTREAM\_VERSION =** “<valid/supported mumps source versions: 5.4.1, 5.5.0, and 5.5.1>”



6. Toggle/Edit the options in step 5 to get:

- a. Debug or Release build
- b. LP64 or ILP64 libs
- c. AOCL or MKL Libs

7. Build the project:

```
cmake --build ninja_build_dir --config Release --target install --verbose
```

8. Run the executable in *ninja\_build\_dir\tests*:

```
mpiexec -n 2 --map-by L3cache --bind-to core Csimple.exe  
mpiexec -n 4 --map-by L3cache --bind-to core amd_aocl.exe sample.mtx 1 1 10
```

---

## Chapter 17 AOCL Tuning Guidelines

---

This section provides tuning recommendations for AOCL.

### 17.1 AOCL-BLAS Thread Control

Application can set the desired number of threads during AOCL-BLAS initialization and runtime as explained below.

#### 17.1.1 AOCL-BLAS Initialization

During AOCL-BLAS initialization, the preferred number of threads by an application in the BLAS routines can be set in multiple ways as follows:

- `bli_thread_set_num_threads(nt)` AOCL-BLAS library API
- Valid value of `BLIS_NUM_THREADS` environment variable
- `omp_set_num_threads(nt)` OpenMP library API
- Valid value of `OMP_NUM_THREADS` environment variable
- If none of these is issued by an application, the number of logical cores would be used by the AOCL-BLAS library as the preferred number of threads

If the number of threads is set in one or more possible ways, the order of precedence for AOCL would be in the above mentioned order.

The following table describes the sample scenarios for setting the number of threads during AOCL-BLAS initialization:

**Table 26. Sample Scenarios - 1**

Sample Pseudo Code for Application	Sample Command Executed	Number of Threads Set During AOCL-BLAS Initialization	Remarks
<pre>int main() {     ///pseudo     code to use OpenMP     API to set number of     threads ///     omp_set_num_threads(     16);     dgemm( );     ///     return 0; }</pre>	\$ BLIS_NUM_THREADS=8 ./my_blis_program	8	BLIS_NUM_THREADS will have the maximum precedence.
	\$ ./my_blis_program	16	BLIS_NUM_THREADS is not set and hence, omp_set_num_threads(16) has taken effect.
	\$ OMP_NUM_THREADS=4 ./my_blis_program	16	BLIS_NUM_THREADS is not set, omp_set_num_threads(16) has taken effect as it has more precedence than OMP_NUM_THREADS.
	\$ BLIS_NUM_THREADS=8 OMP_NUM_THREADS=4 ./my_blis_program	8	BLIS_NUM_THREADS is set to 8, omp_set_num_threads(nt) and OMP_NUM_THREADS do not have any effect.
<pre>int main() {     ///pseudo     code ///     dgemm( );     ///     return 0; }</pre>	\$ BLIS_NUM_THREADS=8 ./my_blis_program	8	BLIS_NUM_THREADS will have the maximum precedence.
	\$ ./my_blis_program	64	BLIS_NUM_THREADS is not set, omp_set_num_threads() is not issued, and OMP_NUM_THREADS is not set, Considering the number of logical cores to be 64, number of threads is 64.
	\$ OMP_NUM_THREADS=4 ./my_blis_program	4	BLIS_NUM_THREADS is not set, omp_set_num_threads() is not issued, and OMP_NUM_THREADS is set to 4.

### 17.1.2 Runtime

Once the number of threads is set during AOCL-BLAS initialization, it will be used in subsequent BLAS routine execution until the application modifies the number of threads (for example, omp\_set\_num\_threads() API) to be used.

The following table describes the sample scenarios for setting the number of threads during runtime:

**Table 27. Sample Scenarios - 2**

Sample Pseudo Code for Application	Sample Command Executed	m Value in Sequence of Execution	Number of Threads for this BLAS Call	Remarks
<pre> int main() {   ///Pseudo code for   sample usage of OpenMP   API to set number of   threads in the   Application during Run   Time/////   do {     if(m &lt; 500)       omp_set_num_threads(8);     if(m &gt;=       500)       omp_set_num_threads(16);      if(m &gt;=       3000)       omp_set_num_threads(32);       dgemm_( );     }     while(test_case_counter-       -)       ///////////     return 0;   } </pre>	\$. /my_blis_program	100	8	Application issued omp_set_num_threads(8)
		500	16	Application issued omp_set_num_threads(16)
		200	8	Application re-issued omp_set_num_threads(8)
		4000	32	Application issued omp_set_num_threads(32)
		1000	16	Application re-issued omp_set_num_threads(16)
		500	16	Application re-issued omp_set_num_threads(16)
		100	8	Application re-issued omp_set_num_threads(8)

### 17.1.2.1 Runtime Thread Control

AOCL-BLAS libraries that are multi-threaded using OpenMP parallelism provide two mechanisms for the users to control the number of threads for AOCL-BLAS functions to use. These are the normal OpenMP mechanisms and AOCL-BLAS specific environment variables and function calls. The AOCL-BLAS specific mechanisms include the option to set the overall number of threads for AOCL-BLAS to use or to set the threading specifically for the different loops within the AOCL-BLAS3 routines (for example, DGEMM). These are called the automatic and the manual ways respectively. For more information, refer to:

<https://github.com/amd/blis/blob/master/docs/Multithreading.md>

The order of precedence used in AOCL-BLAS, where set or called by the user, is as follows:

1. The AOCL-BLAS manual way values set using `bli_thread_set_ways()` by the application.
2. Valid value(s) of any of the `BLIS_*_NT` environment variables.
3. Value set using `bli_thread_set_num_threads(nt)` by the application.
4. Valid value set for the environment variable `BLIS_NUM_THREADS`.
5. `omp_set_num_threads(nt)` issued by the application.
6. Valid value set for the environment variable `OMP_NUM_THREADS`.
7. The default number of threads used by the chosen OpenMP runtime library when `OMP_NUM_THREADS` is not set.

Two other factors may override these settings:

1. OpenMP parallelism at higher level(s) in the code calling AOCL-BLAS, that is, the number of active levels and the level at which the AOCL-BLAS call occurs.
2. The effect of AOCL Dynamic (if enabled), as described in the next section.

**Note:** *AOCL 4.1 has improved support for calling AOCL-BLAS within nested OpenMP parallelism compared to the previous releases. Hence, using the standard OpenMP mechanisms should be sufficient for most of the use cases.*

## 17.2 AOCL Dynamic

The AOCL dynamic feature enables AOCL-BLAS to dynamically change the number of threads.

This feature is enabled by default, however, it can be enabled or disabled at the configuration time using the options `--enable-aocl-dynamic` and `--disable-aocl-dynamic` respectively.

You can also specify the preferred number of threads using the environment variables `BLIS_NUM_THREADS` or `OMP_NUM_THREADS`, `BLIS_NUM_THREADS` takes precedence if both of them are specified.

The following table summarizes how the number of threads is determined based on the status of AOCL Dynamic and the user configuration using the variable `BLIS_NUM_THREADS`:

**Table 28. AOCL Dynamic**

AOCL Dynamic	BLIS_NUM_THREADS	Number of Threads Used by AOCL-BLAS
Disabled	Unset	Number of Cores.
Disabled	Set	BLIS_NUM_THREADS
Enabled <sup>a</sup>	Unset	Number of threads determined by AOCL Dynamic.
Enabled <sup>a</sup>	Set	Minimum of BLIS_NUM_THREADS or the number of threads determined by AOCL.

- a. The AOCL dynamic feature currently supports only DGEMM, DGEMMT, DTRSM, DTRMM, and DSYRK APIs. For the other APIs, the threads selection will be same as when AOCL Dynamic is disabled.

### 17.2.1 Limitations

The AOCL Dynamic feature has the following limitations:

- Support only for OpenMP Threads
- Supports only DGEMM, ZGEMM, DTRSM, ZTRSM, DGEMMT, DSYRK, DTRMM, SGEMV, DSCAL, ZDSCAL, DDOT, and DAXPY APIs
- Specifying the number of threads more than the number of cores may result in deteriorated performance because of over-utilization of cores

## 17.3 AOCL-BLAS DGEMM Multi-thread Tuning

A AOCL-BLAS library can be used on multiple platforms and applications. Multi-threading adds more configuration options at runtime. This section explains the number of threads and CPU affinity settings that can be tuned to get the best performance for your requirements.

### 17.3.1 Library Usage Scenarios

- The application and library are single-threaded:

This is straight forward - no special instructions needed. You can export `BLIS_NUM_THREADS=1` indicating you are running AOCL-BLAS in a single-thread mode. If both `BLIS_NUM_THREADS` and `OMP_NUM_THREADS` are set, the former will take precedence over the later.

- The application is single-threaded and the library is multi-threaded:

You can either use `OMP_NUM_THREADS` or `BLIS_NUM_THREADS` to define the number of threads for the library. However, it is recommend that you use `BLIS_NUM_THREADS`.

Example:

```
$ export BLIS_NUM_THREADS=128 // Here, AOCL-BLAS runs at 128 threads.
```

Apart from setting the number of threads, you must pin the threads to the cores using `GOMP_CPU_AFFINITY` or `numactl` as follows:

```
$ BLIS_NUM_THREADS=128 GOMP_CPU_AFFINITY=0-127 <./application>
```

Or

```
$ BLIS_NUM_THREADS=128 GOMP_CPU_AFFINITY=0-127 numactl --i=all <./application>
$ BLIS_NUM_THREADS=128 numactl -C 0-127 --interleave=all <./test_application.x>
```

**Note:** For the Clang compiler, it is mandatory to use `OMP_PROC_BIND=true` in addition to the thread pinning (if `numactl` is used). For example, for a matrix size of 200 and 32 threads, if you run DGEMM without `OMP_PROC_BIND` settings, the performance would be less. However, if you start using `OMP_PROC_BIND=true`, the performance

*would improve. This problem is not noticed with libgomp using gcc compiler. For the gcc compiler, the processor affinity defined using numactl is sufficient.*

- The application is multi-threaded and the library is running a single-thread:

When the application is running multi-thread and number of threads are set using OMP\_NUM\_THREADS, it is mandatory to set BLIS\_NUM\_THREADS to one. Otherwise, AOCL-BLAS will run in multi-threaded mode with the number of threads equal to OMP\_NUM\_THREADS. This may result in a poor performance.

- The application and library are both multi-threaded:

This is a typical scenario of nested parallelism. To individually control the threading at application and at the AOCL-BLAS library level, use both OMP\_NUM\_THREADS and BLIS\_NUM\_THREADS.

- The number of threads launched by the application is OMP\_NUM\_THREADS.
- Each application thread spawns BLIS\_NUM\_THREADS threads.
- To get a better performance, ensure that Number of Physical Cores = OMP\_NUM\_THREADS \* BLIS\_NUM\_THREADS.

Thread pinning for the application and the library can be done using OMP\_PROC\_BIND:

```
$ OMP_NUM_THREADS=4 BLIS_NUM_THREADS=8 OMP_PROC_BIND=spread,close <./application>
```

#### **OMP\_PROC\_BIND=spread,close**

At an outer level, the threads are spread and at the inner level, the threads are scheduled closer to their master threads. This scenario is useful for a nested parallelism, where the application is running at say OMP\_NUM\_THREADS and each thread is calling multi-threaded AOCL-BLAS.

## **17.3.2 Architecture Specific Tuning**

### **17.3.2.1 2<sup>nd</sup> and 3<sup>rd</sup> Gen AMD EPYC™ Processors**

To achieve the best DGEMM multi-thread performance on 2<sup>nd</sup> Gen AMD EPYC™ processors (codenamed "Rome") and 3<sup>rd</sup> Gen AMD EPYC™ processors (codenamed "Milan"), execute one of the following commands:

#### **Thread Size up to 16 (< 16)**

```
OMP_PROC_BIND=spread OMP_NUM_THREADS=<NT> ./test_gemm_blis.x
```

#### **Thread Size above 16 (>= 16)**

```
OMP_PROC_BIND=spread OMP_NUM_THREADS=<NT> numactl --interleave=all ./test_gemm_blis.x
```

### **17.3.2.2 1<sup>st</sup> Gen AMD EPYC™ Processors**

To achieve the best DGEMM multi-thread performance on the 1<sup>st</sup> Gen AMD EPYC™ processors (codenamed "Naples"), complete the following steps:

The header file *bli\_family\_zen.h* in the AOCL-BLAS source directory `\\blis\config\zen` defines certain macros that help control the block sizes used by AOCL-BLAS.

The required tuning settings vary depending on the number threads that the application linked to AOCL-BLAS runs.

### Thread Size upto 16 (< 16)

1. Enable the macro `BLIS_ENABLE_ZEN_BLOCK_SIZES` in the file *bli\_family\_zen.h*.
2. Compile AOCL-BLAS with multi-thread option as mentioned in “Multi-thread AOCL-BLAS” on page 25.
3. Link the generated AOCL-BLAS library to your application and execute it.
4. Run the application:

```
OMP_PROC_BIND=spread BLIS_NUM_THREADS=<NT> ./test_gemm_blis.x
```

### Thread Size above 16 (>= 16)

1. Disable the macro `BLIS_ENABLE_ZEN_BLOCK_SIZES` in the file *bli\_family\_zen.h*.
2. Compile AOCL-BLAS with the multi-thread option as mentioned in “Multi-thread AOCL-BLAS” on page 25.
3. Link the generated AOCL-BLAS library to your application.
4. Set the following OpenMP and memory interleaving environment settings:

```
OMP_PROC_BIND=spread
BLIS_NUM_THREADS = x      // x> 16
numactl --interleave=all
```

5. Run the application.

Example:

```
OMP_PROC_BIND=spread BLIS_NUM_THREADS=<NT> numactl --interleave=all ./test_gemm_blis.x
```

## 17.4 AOCL-BLAS DGEMM Block-size Tuning

AOCL-BLAS DGEMM performance is largely impacted by the block sizes used by AOCL-BLAS. A matrix multiplication of large m, n, and k dimensions is partitioned into sub-problems of the specified block sizes.

Many HPC, scientific applications, and benchmarks run on high-end cluster of machines, each with multiple cores. They run programs with multiple instances through Message Passing Interface (MPI) based APIs or separate instances of each program. Depending on whether the application using AOCL-BLAS is running in multi-instance mode or single instance, the specified block sizes will have an impact on the overall performance.

The default values for the block size in AOCL-BLAS GitHub repository (<https://github.com/amd/blis>) is set to extract the best performance for such HPC applications/benchmarks, which use single-threaded AOCL-BLAS and run in multi-instance mode on AMD EPYC™ AMD “Zen” core



processors. However, if your application runs as a single instance, the block sizes for an optimal performance would vary.

The following settings will help you choose the optimal values for the block sizes based on the way the application is run:

## 2nd Gen AMD EPYC™ Processors (codenamed "Rome")

1. Open the file *bli\_family\_zen2.h* in the AOCL-BLAS source:

```
$ cd "config/zen2/ bli_family_zen2.h"
```

2. For applications/benchmarks running in multi-instance mode and using multi-threaded AOCL-BLAS, ensure that the macro AOCL\_BLIS\_MULTIINSTANCE is set to 0. As of AOCL 2.x release, this is the default setting. The HPL benchmark is found to generate better performance numbers using the following setting for multi-threaded AOCL-BLAS:

```
#define AOCL_BLIS_MULTIINSTANCE 0
```

## 1st Gen AMD EPYC™ Processors (codenamed "Naples")

1. Open the file *bli\_cntx\_init\_zen.c* under the AOCL-BLAS source:

```
$ cd "config/zen/bli_family_zen.h"
```

2. Ensure the macro, BLIS\_ENABLE\_ZEN\_BLOCK\_SIZES is defined:

```
#define BLIS_ENABLE_ZEN_BLOCK_SIZES
```

### Multi-instance Mode

For applications/benchmarks running in multi-instance mode, ensure that the macro BLIS\_ENABLE\_SINGLE\_INSTANCE\_BLOCK\_SIZES is set to 0. As of AOCL 2.x release, following is the default setting:

```
#define BLIS_ENABLE_SINGLE_INSTANCE_BLOCK_SIZES 0
```

The optimal block sizes for this mode on AMD EPYC™ are defined in the file *config/zen/bli\_cntx\_init\_zen.c*:

```
bli_blksize_init_easy( &blksize[ BLIS_MC ], 144, 240, 144, 72 );
bli_blksize_init_easy( &blksize[ BLIS_KC ], 256, 512, 256, 256 );
bli_blksize_init_easy( &blksize[ BLIS_NC ], 4080, 2040, 4080, 4080 );
```

### Single-instance Mode

For the applications running as a single instance, ensure that the macro BLIS\_ENABLE\_SINGLE\_INSTANCE\_BLOCK\_SIZES is set to 1:

```
#define BLIS_ENABLE_SINGLE_INSTANCE_BLOCK_SIZES 1
```

The optimal block sizes for this mode on AMD EPYC™ are defined in the file *config/zen/bli\_cntx\_init\_zen.c*:

```
bli_blksize_init_easy( &blksize[ BLIS_MC ], 144, 510, 144, 72 );
bli_blksize_init_easy( &blksize[ BLIS_KC ], 256, 1024, 256, 256 );
bli_blksize_init_easy( &blksize[ BLIS_NC ], 4080, 4080, 4080, 4080 );
```

## 17.5 Performance Suggestions for Skinny Matrices

AOCL-BLAS provides a selective packing for GEMM when one or two-dimensions of a matrix is exceedingly small. Selective packing is only applicable when **sup** is enabled. For an optimal performance:

```
C = beta*C + alpha*A*B
Dimension (Dim) of A - m x k          Dim(B) - k x n          Dim(c) - m x n
Assume row-major.
IF m >> n
$BLIS_PACK_A=1 ./test_gemm_blis.x - will give a better performance.
IF m << n
$BLIS_PACK_B=1 ./test_gemm_blis.x - will give a better performance.
```

## 17.6 AOCL-LAPACK Multi-threading

From AOCL 4.0 release, AOCL-LAPACK supports multi-threading using OpenMP in selected APIs. This feature is enabled by default when AOCL-LAPACK is compiled with `--enable-amd-flags` or `--enable-amd-aocc-flags`. However, you can disable multi-threading by setting `--enable-multithreading=no`.

The selected LAPACK interface APIs that support multi-threading automatically choose optimal number of threads. However, you can explicitly set the number of threads through the environment variable or OpenMP runtime APIs. In such a scenario, the number of threads is selected as follows:

Thread Criteria	Threads Used by API
User specified threads > AOCL-LAPACK computed optimal threads	AOCL-LAPACK computed optimal threads
User specified threads < AOCL-LAPACK computed optimal threads	User specified threads

## 17.7 AOCL-FFTW Tuning Guidelines

Following are the tuning guidelines to get the best performance out of AMD optimized FFTW:

- Use the configure option `--enable-amd-opt` to build the targeted library. This option enables all the improvements and optimizations meant for AMD EPYC™ CPUs.

This is the mandatory master optimization switch that must be set for enabling any other optional configure options, such as:

```
- --enable-amd-mpifft
- --enable-amd-mpi-vader-limit
- --enable-amd-trans
- --enable-amd-fast-planner
- --enable-amd-top-n-planner
- --enable-amd-app-opt
- --enable-dynamic-dispatcher
```

- When enabling the AMD CPU specific improvements with the configure option `--enable-amd-opt`, do not use the configure option `--enable-generic-simd128` or `--enable-generic-simd256`.
- An optional configure option `--enable-amd-trans` is provided and it may benefit the performance of transpose operations in the case of very large FFT problem sizes. This feature is to be used only when running in single-thread and single instance mode.
- Use the configure option `--enable-amd-mpifft` to enable MPI FFT related optimizations. This is provided as an optional parameter and will benefit most of the MPI problem types and sizes.
- An optional configure option `--enable-amd-mpi-vader-limit` that controls enabling of AMD's new MPI transpose algorithms is supported. When using this configure option, you must set `--mca btl_vader_eager_limit` appropriately (current preference is 65536) in the MPIRUN command.
- You can enable AMD optimized fast planner using the optional configure option `--enable-amd-fast-planner`. You can use this option to reduce the planning time without much trade-off in the performance. It is supported for single and double precisions.
- To minimize single-threaded run-to-run variations, you can enable the planner feature Top N planner using configure option `--enable-amd-top-n-planner`. It works by employing WISDOM feature to generate and reuse a set of top N plans for the given size (wherein the value of N is currently set to 3). It is supported for only single-threaded execution runs.
- For best performance, use the `PATIENT` planner flag of FFTW.

A sample running of FFTW bench test application with `PATIENT` planner flag is as follows:

```
$ ./bench -opatient -s icf65536
```

Where, `-s` option is for speed/performance run and `icf` options stand for in-place, complex data-type, and forward transform.

- When configured with `--enable-openmp` and running multi-threaded test, set the OpenMP variables as:

```
set OMP_PROC_BIND=TRUE
OMP_PLACES=cores
```

Then, run the test bench executable binary using `numactl` as follows:

```
numactl --interleave=0,1,2,3 ./bench -opatient -onthreads=64 -s icf65536
```

Where, `numactl --interleave=0,1,2,3` sets the memory interleave policy on nodes 0, 1, 2, and 3.

- When running MPI FFTW test, set the appropriate MPI mapping, binding, and rank options.

For example, to run 64 MPI rank FFTW on a 64-core AMD EPYC™ processor, use:

```
mpirun --map-by core --rank-by core --bind-to core -np 64 ./mpi-bench -opatient -s icf65536
```

- Use the configure option `--enable-amd-app-opt` to enable AMD's application optimization layer in AOCL-FFTW to help uplift performance of various HPC and scientific applications. For more information, refer "AOCL-FFTW" on page 168.
- To build a single portable optimized library that can run on a wide range of CPU architectures, a dynamic dispatcher feature is implemented. Use `--enable-dynamic-dispatcher` configure option to enable this feature for Linux-based systems. The set of x86 CPUs on which the single portable library can work depends on the highest level of CPU SIMD instruction set with which it is configured.

## Chapter 18 Support

---

For support options, the latest documentation, and downloads refer to AMD Developer Central (<https://www.amd.com/en/developer/aocl.html>).

## Chapter 19 References

---

The following URLs have been used as references for this document:

- <https://www.amd.com/en/developer/aocl.html>
- <http://www.netlib.org>
- <http://www.netlib.org/benchmark/hpl/>
- <https://dl.acm.org/citation.cfm?id=2764454>
- <https://github.com/flame/blis>
- <http://fftw.org/>
- <http://mumps-solver.org/>
- <https://spack.io/>

# Appendix

---

## Check AMD Server Processor Architecture

### On Linux

To identify your AMD processor's generation, perform the following steps on Linux:

1. Run the command:

```
$ lscpu
```

2. Check the values of CPU family and Model fields:

- a. For 1<sup>st</sup> Gen AMD EPYC™ Processors (codenamed “Naples”), CPU Core AMD “Zen”
  - CPU Family: 23
  - Model: Values in the range <1 – 47>
- b. For 2<sup>nd</sup> Gen AMD EPYC™ Processors (codenamed “Rome”), CPU Core AMD “Zen2”
  - CPU Family: 23
  - Model: Values in the range <48 – 63>
- c. For 3<sup>rd</sup> Gen AMD EPYC™ Processors (codenamed “Milan”), CPU Core AMD “Zen3”
  - CPU Family: 25
  - Model: Values in the range <1 – 15>
- d. For 4<sup>th</sup> Gen AMD EPYC™ Processors (codenamed “Genoa”), CPU Core AMD “Zen4”
  - CPU Family: 25
  - Model: Values in the range <16–31, 96–111, 120–123, 160–175>

### On Windows

To identify your AMD processor's generation, perform the following steps on Windows:

1. Run the command in Windows **Command Prompt**:

```
wmic cpu get caption
```

2. Check the values of CPU family and Model fields:

- a. For 1<sup>st</sup> Gen AMD EPYC™ Processors (codenamed “Naples”), CPU Core AMD “Zen”
  - CPU Family: 23
  - Model: Values in the range <1 – 47>
- b. For 2<sup>nd</sup> Gen AMD EPYC™ Processors (codenamed “Rome”), CPU Core AMD “Zen2”
  - CPU Family: 23
  - Model: Values in the range <48 – 63>

- c. For 3<sup>rd</sup> Gen AMD EPYC™ Processors (codenamed “Milan”), CPU Core AMD “Zen3”
  - CPU Family: 25
  - Model: Values in the range <1 – 15>
- d. For 4<sup>th</sup> Gen AMD EPYC™ Processors (codenamed “Genoa”), CPU Core AMD “Zen4”
  - CPU Family: 25
  - Model: Values in the range <16–31, 96-111, 120-123, 160-175>

## Application Notes

### AOCL-BLAS

If you prefer to build the application or the test suite executable with the pre-built static library (from the package) on Windows, both the instances of "#define BLIS\_ENABLE\_SHARED" must be commented out in the header file *blis.h*.

### AOCL-FFTW

- Quad precision is supported in AOCL-FFTW using the AOCC v2.2 compiler (AMD clang version 10 onwards).
- Feature **AMD application optimization layer** has been introduced in AOCL-FFTW to uplift the performance of various HPC and scientific applications.
  - The configure option `--enable-amd-app-opt` enables this optimization layer and must be used with the master optimization configure switch `--enable-amd-opt` mandatorily.
  - This optimization layer is supported for complex and real (r2c and c2r) DFT problem types in double and single precisions.
  - Not supported for MPI FFTs, real r2r DFT problem types, Quad or Long double precisions, and split array format.