

AMD

AOCL User Guide

Publication # **57404**

Revision # **4.2**

Issue Date **February 2024**

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

Revision History	12
Chapter 1 Introduction	13
1.1 Feature Support Matrix	14
Chapter 2 Validation Matrix	17
2.1 Operating Systems	17
2.2 Compilers	17
2.3 Library	17
2.4 Message Passing Interface (MPI)	17
2.5 Programming Language	17
2.6 Build Utilities	18
Chapter 3 Installing AOCL	19
3.1 Building from Source	19
3.2 Installing AOCL Binary Packages	19
3.2.1 Using Master Package	19
3.2.2 Using Library Package	21
3.2.3 Using Debian and RPM Packages	21
3.2.4 Using Windows Packages	23
Chapter 4 AOCL-BLAS	25
4.1 Installation on Linux	25
4.1.1 Build AOCL-BLAS from Source	25
4.1.2 Using Pre-built Binaries	27
4.2 Application Development Using AOCL-BLAS	27
4.2.1 API Compatibility Layers (Calling AOCL-BLAS)	27
4.2.2 API Compatibility - Advance Options	29
4.2.3 Linking Application with AOCL-BLAS	29
4.2.4 AOCL-BLAS Usage in Fortran	30
4.2.5 AOCL-BLAS Usage in C	32
4.3 Migrating/Porting	36
4.4 Using AOCL-BLAS Library Features	37

4.4.1	Dynamic Dispatch	37
4.4.2	AOCL-BLAS - Running the Test Suite	40
4.4.3	Testing/Benchmarking	41
4.4.4	AOCL-BLAS Utility APIs	43
4.5	Debugging and Troubleshooting	44
4.5.1	Error Handling in AOCL-BLAS	44
4.5.2	Debugging Build Using GDB	45
4.5.3	Viewing Logs	46
4.5.4	Checking AOCL-BLAS Operation Progress	50
4.6	Build AOCL-BLAS from Source on Windows	52
4.6.1	Building AOCL-BLAS using GUI	53
4.6.2	Building AOCL-BLAS using Command-line Arguments	58
4.6.3	Packaging AOCL -BLAS	58
4.6.4	Building and Running the Test Suite	58
4.7	LPGEMM in AOCL-BLAS	59
4.7.1	Add-on in AOCL-BLAS	59
4.7.2	API Naming and Arguments	59
4.7.3	Post-operations	60
4.7.4	APIs and Post-ops in aocl_gemm	61
4.7.5	Enabling aocl_gemm Add-on	62
4.7.6	Sample Application 1	63
4.7.7	Sample Application 2	65
Chapter 5	AOCL-LAPACK	68
5.1	Installing on Linux	68
5.1.1	Building AOCL-LAPACK from Source	68
5.1.2	Using Pre-built Libraries	73
5.2	Usage on Linux	73
5.2.1	Use by Applications	74
5.3	Building AOCL-LAPACK from Source on Windows	75
5.3.1	Building AOCL-LAPACK Using GUI	75
5.3.2	Building AOCL-LAPACK using Command-line Arguments	78

5.3.3	Building and Running Test Suite	78
5.4	Checking AOCL-LAPACK Operation Progress	79
Chapter 6	AOCL-FFTW	81
6.1	Installing	81
6.1.1	Building AOCL-FFTW from Source on Linux	81
6.1.2	Building AOCL-FFTW from Source on Windows	83
6.1.3	Using Pre-built Libraries	87
6.2	Usage	87
6.2.1	Sample Programs for Single-threaded and Multi-threaded FFTW	87
6.2.2	Sample Programs for MPI FFTW	88
6.2.3	Additional Options	88
Chapter 7	AOCL-LibM	89
7.1	Library Contents	89
7.1.1	Scalar Functions	89
7.1.2	Fast Scalar and Vector Variants	90
7.2	Installation	93
7.2.1	Installing the Pre-Built Binaries on Linux	93
7.2.2	Building AOCL-LibM on Linux	93
7.2.3	Building AOCL-LibM on Windows	94
7.3	Using AOCL-LibM	96
Chapter 8	AOCL-ScaLAPACK	98
8.1	Installation	98
8.1.1	Building AOCL-ScaLAPACK from Source on Linux	99
8.1.2	Using Pre-built Libraries	102
8.2	Usage	102
8.3	Building AOCL-ScaLAPACK from Source on Windows	103
8.3.1	Building AOCL-ScaLAPACK Using GUI	103
8.3.2	Building AOCL-ScaLAPACK using Command-line Arguments	104
8.3.3	Building and Running the Individual Tests	105
8.4	Checking AOCL-ScaLAPACK Operation Progress	105
8.5	Additional Features	106

Chapter 9	AOCL-RNG	108
9.1	Installation	108
9.2	Using AOCL-RNG Library on Linux	109
9.3	Using AOCL-RNG Library on Windows	109
Chapter 10	AOCL-SecureRNG	110
10.1	Installation	110
10.2	Usage	110
10.3	Using AOCL-SecureRNG Library on Windows	111
Chapter 11	AOCL-Sparse	113
11.1	Installation	114
11.1.1	Building AOCL-Sparse from Source on Linux	114
11.1.2	Building AOCL-Sparse from Source on Windows	116
11.1.3	Using Pre-built Libraries	120
11.2	Usage	120
11.2.1	Use by Applications on Linux	120
11.2.2	Use by Application on Windows	121
11.2.3	Performance Benchmarking on Linux	122
11.2.4	Performance Benchmarking on Windows	123
11.2.5	Running the Test Suite	124
Chapter 12	AOCL-LibMem	125
12.1	Building AOCL-LibMem for Linux	125
12.2	Running an Application	126
12.3	Running an Application with Tunables	126
12.3.1	Default State	127
12.3.2	Tuned State	127
Chapter 13	AOCL-Cryptography	130
13.1	Requirements	131
13.2	Installation	131
13.2.1	Building AOCL-Cryptography from Source on Linux	131
13.2.2	Building AOCL-Cryptography from Source on Windows	133
13.3	Using AOCL-Cryptography in a Sample Application	134

13.3.1	Compiling and Running Examples	134
13.3.2	AOCL-Cryptography Library Provider for OpenSSL	135
13.3.3	Integrating AOCL Libraries with Applications that Use IPP	135
Chapter 14	AOCL-Compression	136
14.1	Installation	136
14.1.1	Using Pre-built Libraries	136
14.1.2	Building from Source	137
14.2	Running AOCL-Compression Test Bench on Linux	137
14.3	Running AOCL-Compression Test Bench on Windows	139
14.4	API Reference	139
14.4.1	Unified Standardized API Set	139
14.4.2	Interface Data Structures	139
14.4.3	Library Return Error Codes	141
14.4.4	Multi-threaded API Set	141
14.4.5	Native APIs	141
14.4.6	Example Program	144
14.5	Optional Optimization Options	148
Chapter 15	AOCL-Utils	150
15.1	Requirements	151
15.2	Clone and Build the AOCL-Utils Library	151
15.3	Using AOCL-Utils	151
15.3.1	C API Example	152
15.3.2	C++ API Example	152
15.3.3	Building on Windows	152
15.3.4	Building on Linux	153
15.3.5	Output	154
15.3.6	Integrate with Other Libraries/Applications	154
Chapter 16	AOCL Tuning Guidelines	155
16.1	AOCL-BLAS Thread Control	155
16.1.1	AOCL-BLAS Initialization	155
16.1.2	Runtime	156

16.2	AOCL Dynamic	158
16.2.1	Limitations	159
16.3	AOCL-BLAS DGEMM Multi-thread Tuning	159
16.3.1	Library Usage Scenarios	159
16.3.2	Architecture Specific Tuning	160
16.4	AOCL-BLAS DGEMM Block-size Tuning	161
16.5	Performance Suggestions for Skinny Matrices	163
16.6	AOCL-LAPACK Multi-threading	163
16.7	AOCL-FFTW Tuning Guidelines	164
Chapter 17	Support	166
Chapter 18	References	167
Appendix	168
	Check AMD Server Processor Architecture	168
	On Linux	168
	On Windows	168
	Application Notes	169
	AOCL-BLAS.....	169
	AOCL-FFTW.....	169

List of Tables

Table 1.	AOCL Feature Support Matrix - 1	14
Table 2.	AOCL Feature Support Matrix - 2	15
Table 3.	install.sh Script Options	20
Table 4.	AOCL-BLAS API Compatibility Layers	28
Table 5.	AOCL-BLAS API Compatibility - Advance Options	29
Table 6.	AOCL-BLAS Application - Link Options	29
Table 7.	Porting to AOCL-BLAS	37
Table 8.	AOCL-BLAS Utility APIs	43
Table 9.	AOCL-BLAS - Error Handlers	44
Table 10.	Callback Parameters	51
Table 11.	CMake Config Options	55
Table 12.	Required Architecture Features and APIs	61
Table 13.	GEMM API Supported Post-ops.	61
Table 14.	Utility APIs in aocl_gemm Add-on	62
Table 15.	AOCL-LAPACK Config Options	75
Table 16.	AOCL-LAPACK Progress Feature Callback Function Parameters	79
Table 17.	AOCL-FFTW Config Options	84
Table 18.	AOCL-ScaLAPACK CMake Parameter List	98
Table 19.	Compiler and Type of Library	100
Table 20.	AOCL-ScaLAPACK Progress Feature Callback Function Parameters	106
Table 21.	Additional Features	106
Table 22.	Compiler and Library Type.	115
Table 23.	AOCL-Sparse - CMake Build Options	115
Table 24.	Application Implementations	127
Table 25.	Sample Threshold Settings	129
Table 26.	AOCL-Cryptography - Linux Options	132
Table 27.	AOCL-Cryptography - Windows Options	133
Table 28.	Optional Optimization Options	148
Table 29.	Sample Scenarios - 1	156

Table 30.	Sample Scenarios - 2.....	157
Table 31.	AOCL Dynamic	158

List of Figures

Figure 1.	Sample Run of Function Call Tracing	48
Figure 2.	Sample Run with Debug Logs Enabled	49
Figure 3.	Debug Logs Showing Input Values of GEMM	50
Figure 4.	Microsoft Visual Studio Prerequisites	52
Figure 5.	CMake Source and Build Folders.....	53
Figure 6.	Set Generator and Compiler	54
Figure 7.	CMake Configure and Generate Project Settings.....	57
Figure 8.	AOCL-LAPACK CMake Configurations	77
Figure 9.	AOCL-FFTW CMake Config Options.....	86
Figure 10.	AOCL-ScaLAPACK CMake Options	104
Figure 12.	AOCL-Sparse CMake Config Options.....	118

Revision History

Date	Revision	Description
February 2024	4.2	<ul style="list-style-type: none">• Re-organized Chapter 11• Release specific updates and general edits
August 2023	4.1	<ul style="list-style-type: none">• Added Chapter 15• Added sections 4.7, 8.5, 11.5, 14.1.1, and 14.1.2
November 2022	4.0	<ul style="list-style-type: none">• Added sections 9.3, 10.3, 16.1.2.1, and 16.6• Updated section 4.4.1.3• Added Chapter 14• Removed the chapter AOCL-Spack recipes
July 2022	3.2	<ul style="list-style-type: none">• Added chapters 12 and 13, sections 5.4, 8.4, and 16.1• Added Multi-thread support information in chapter 11
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** is a software library containing elementary 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
AOCL-BLAS	Yes	Yes	Yes	Single, Double, Complex, Double Complex, Mixed Precision, and Low Precision (INT16, INT8, UINT8, BFLOAT16, and so on) <i>Note: Currently supported only for GEMM API.</i>
AOCL-LAPACK	No	Partially (requires AVX2 support)	Not applicable	Single, Double, Complex, Double Complex
AOCL-FFTW	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
AOCL-LibM	Yes	Yes	Yes	Single, Double, Complex, Double Complex
AOCL-Sparse	Partial (for SpMV)	Partial (for SpMV)	Yes	Single, Double, Complex, Complex, Double Complex
AOCL-Cryptography	Yes	Yes, GCC and AOCC on Linux; Clang on Windows.	Not applicable	Not applicable
AOCL-Compression	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

Table 1. AOCL Feature Support Matrix - 1

Library\Feature	AVX512	Dynamic Dispatcher	Vector	Precision
AOCL-RNG	Partial	Yes	Not applicable	Single, Double
AOCL-SecureRNG	Not applicable	Not applicable	Not applicable	Not applicable
AOCL-ScaLAPACK	Dependent on the underlying BLAS and LAPACK libraries	Dependent on the underlying BLAS and LAPACK libraries	Not applicable	Single, Double, Complex, Double Complex
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	Partial (for LZ4, Snappy, ZLIB, and ZSTD)	No
AOCL-RNG	Yes	Yes	No	No

Table 2. AOCL Feature Support Matrix - 2

Library\Feature	glibc Dependency	Single-threaded	Multi-threaded	MPI
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 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, AOCL-Compression, AOCL-Cryptography, and AOCL-Utills 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.

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

Chapter 2 Validation Matrix

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[®] 20.04 LTS and 22.04 LTS
- Red Hat[®] Enterprise Linux[®] (RHEL) 9.0 and 8.6
- SUSE Linux Enterprise Server (SLES) 15 SP3
- Windows Server 2019
- Windows[®] 10
- Windows 11 Pro

2.2 Compilers

- GCC 12.2 and 13.1
- AOCC 4.1 and 4.2
- LLVM[™] 15 and 16

2.3 Library

- glibc 2.28 and 2.35
- OpenSSL 3.0.0 through 3.0.5

2.4 Message Passing Interface (MPI)

- Linux Open MPI 4.1.5
- Windows Intel[®] MPI (2021.8.0 “version 3.1”)

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)
- SCons 4.6.0

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 are 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/zen-software-studio/applications/spack/spack-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>-4.2.0.tar.gz>` to untar the package.

The installer file *install.sh* is available in `aocl-linux-<compiler>-4.2.0`.

- Run `./install.sh` to install the AOCL package (all libraries) to the default `INSTALL_PATH`: `/home/<username>/aocl/4.2.0/<compiler>`, where the compiler value is `aocc` or `gcc`.

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

Table 3. install.sh Script Options

Option	Description
-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 respectively in the following paths:
 - `/INSTALL_PATH/lib_LP64`
 - `/INSTALL_PATH/include_LP64`
 - ILP64 libraries and header files are installed respectively in the following paths:
 - `/INSTALL_PATH/lib_ILP64`
 - `/INSTALL_PATH/include_ILP64`

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 4.2 Debian package to the target machine.

```
$ dpkg -c aocl-linux-gcc-4.2.0_1_amd64.deb
```

3. Install the package.

```
$ sudo dpkg -i aocl-linux-gcc-4.2.0_1_amd64.deb
Or
$ sudo apt install ./aocl-linux-gcc-4.2.0_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-4.2.0
```

5. List the contents of the package.

```
$ dpkg -L aocl-linux-gcc-4.2.0
```

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>-4.2.0/<compiler>`
- Compiler: `aocc` or `gcc`

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

`/opt/AMD/aocl/aocl-linux-aocc-4.2.0/aocc`

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

```
cd /opt/AMD/aocl/aocl-linux-<compiler>-4.2.0/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-4.2.0
or
$ sudo apt remove aocl-linux-gcc-4.2.0
```

Installing RPM Package

Complete the following steps to install the AOCL RPM package:

1. Download the AOCL 4.2 RPM package to the target machine.
2. Install the package.

```
$ sudo rpm -ivh aocl-linux-gcc-4.2.0-1.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-4.2.0.x86_64
```

4. List the contents of the package.

```
$ rpm -ql aocl-linux-gcc-4.2.0
```

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>-4.2.0/<compiler>`
- Compiler: `aocc` or `gcc`

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

`/opt/AMD/aocl/aocl-linux-aocc-4.2.0/aocc`

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

```
cd /opt/AMD/aocl/aocl-linux-<compiler>-4.2.0/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-4.2.0
```

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 1st Gen AMD EPYC™ (code name "Naples") processor, the **zen** configuration will be auto-selected. For a build machine using the 2nd 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 3rd Gen AMD EPYC™ processor (code name "Milan"). From AOCL-BLAS 4.0 forward, **zen4** will be auto-selected for the 4th 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 37.

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 indicates one of the options in {openmp, no}. "no" option implies 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 19.

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 BLAS/CBLAS standard enables 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 4. AOCL-BLAS API Compatibility Layers

API Compatibility Layer	Header Files	Configuration Option	Usages
BLAS (Fortran)	Not applicable	--enable-blas --disable-blas	Use this option when calling AOCL-BLAS from Fortran applications. API Name Format: DGEMM
BLAS (C)	blis.h	--enable-blas --disable-blas	Use this option when calling AOCL-BLAS from C application using BLAS type parameters. API Name Format: dgemm_
CBLAS	cblas.h	--enable-cblas (Implies --enable-blas) --disable-cblas	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 5. AOCL-BLAS API Compatibility - Advance Options

Feature	Configuration Option	Usages
ILP64 Support	<code>--blas-int-size=SIZE</code>	<p>This option can be used to specify the integer types used in external BLAS/CBLAS interfaces.</p> <p>Accepted Values: ILP64 - SIZE = 64 LP64 - SIZE = 32 (Default)</p>
Complex Number return handling	<code>--complex-return=gnu intel</code>	<p>The complex numbers can be returned through registers or the hidden parameter.</p> <p>Based on the way application is calling the API, the library must be configured to match the return value receptions.</p> <p>gnu = return complex values through registers intel = return complex values through hidden parameter.</p> <p>For more information and example, refer “Returning Complex Numbers” on page 36.</p>

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 6. AOCL-BLAS Application - Link Options

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

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

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 cdotc, cdotu, zdotc, and zdotu 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 7. Porting to AOCL-BLAS

	MKL	OpenBLAS	AOCL-BLAS	
			Single-threaded	Multi-threaded
Header File	mkl.h	cblas.h	blis.h/cblas.h	blis.h/cblas.h
Link Options	-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 25.

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 either the environment variable **AOCL_ENABLE_INSTRUCTIONS** or **BLIS_ARCH_TYPE** as follows:

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

or

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

Where, value = {avx512, avx2, zen4, zen3, zen2, zen, generic}

You must note the following:

- The code path names are not case sensitive but the environment variable names are.
- In AOCL-BLAS builds with configuration **amdzen**, **avx512** is an alias for **zen4** and **avx2** is an alias for **zen3**.
- **AOCL_ENABLE_INSTRUCTIONS** is intended to become the standard option for controlling dynamic dispatch (where supported) across all the AOCL components.
- **BLIS_ARCH_TYPE** is specific to the BLIS code used in AOCL-BLAS.
- If both are specified, **BLIS_ARCH_TYPE** takes precedence and **AOCL_ENABLE_INSTRUCTIONS** is ignored by AOCL-BLAS.
- The operation of **AOCL_ENABLE_INSTRUCTIONS** and **BLIS_ARCH_TYPE** are slightly different:
 - If **AOCL_ENABLE_INSTRUCTIONS** is in operation, AOCL-BLAS will check if the instruction set required by the code path selected is supported by the processor. If not, it will try to step down to another code path that is supported (from AVX512 > AVX2 > generic). In other words, **AOCL_ENABLE_INSTRUCTIONS** should be used to restrict a processor to an earlier instruction set, rather than try to force a later one on an older processor.
 - By contrast, if **BLIS_ARCH_TYPE** is in operation, that code path will be used irrespective of the compatibility with the processor.

- 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. This applies to both `AOCL_ENABLE_INSTRUCTIONS` and `BLIS_ARCH_TYPE`.
 - 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 applies when only when `BLIS_ARCH_TYPE` is used; executing the illegal instruction 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
```

Setting either of these environment variables makes sense only when using a build of AOCL-BLAS that includes multiple code paths.

Thus, `AOCL_ENABLE_INSTRUCTIONS` and `BLIS_ARCH_TYPE` are disabled by default in all the builds containing only a single code path.

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.

From AOCL-BLAS 4.2, 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, 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-bblastest.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 GEMM benchmark 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 input parameters:

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 macro at the start of the file:

```
#define FILE_IN_OUT
```

Building Test Driver

Execute the following commands to build the test driver:

```
$ cd tests  
$ make -j 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 lda ldb ldc`. 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`
- `lda` is leading dimension of matrix A
- `ldb` is leading dimension of matrix B
- `ldc` is leading dimension of matrix C

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

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

- `lda` $\geq m$
- `ldb` $\geq k$
- `ldc` $\geq m$

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 8. AOCL-BLAS Utility APIs

API	Usages
bli_info_get_version_str()	Returns the version string in the form of “AOCL-BLAS 4.2.0 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_info_get_info_value()	Returns the value of INFO from the previous call by this user thread to a BLAS2 or BLAS3 routine. For more information, refer to section 4.5.1.
bli_thread_get_num_threads() ¹	Returns the default number of threads used for the subsequent BLAS calls.
bli_thread_set_num_threads(dim_t n_threads) ¹	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) ¹	Sets the number of threads for different levels of parallelization as per GotoBLAS five loops architecture.
Notes: 1. Refer https://github.com/amd/blis/blob/master/docs/Multithreading.md#specifying-multithreading	

4.5 Debugging and Troubleshooting

4.5.1 Error Handling in AOCL-BLAS

The original Netlib BLAS defined an error handling function XERBLA, which is called within BLAS2 and BLAS3 routines if an incorrect input argument is detected. Only incorrect matrix, vector sizes, and options for specifying transpose matrix, upper or lower in a symmetric matrix, and so on can be detected. BLAS does not detect extreme values (such as Inf or NaNs) within the supplied matrices and vectors, it is the user's responsibility to check for these if required.

The functionality of Netlib's XERBLA is to print a message to standard output and stop execution of the process. Stopping is extremely unhelpful in many applications and usage scenarios. Thus, AOCL-BLAS, in common with other similar libraries, has traditionally disabled the stop statement. In AOCL 4.2, the functionality of AOCL-BLAS has been enhanced to give users more choice over both stopping the application on error and printing a message on error. The choices are specified by setting each of the environment variables `BLIS_STOP_ON_ERROR` and `BLIS_PRINT_ON_ERROR` to 0 or 1 to respectively disable or enable the functionality. The default values for each are:

Table 9. AOCL-BLAS - Error Handlers

Environment Variable	Default Value
<code>BLIS_STOP_ON_ERROR</code>	0
<code>BLIS_PRINT_ON_ERROR</code>	1

When the stop on error is disabled, no error code is passed back to the user application through the BLAS interface arguments, unlike the `INFO` argument used in LAPACK routines. Therefore, AOCL-BLAS has also added an extra function to return the value of `INFO` from the previous call to a BLAS routine made by the same thread. The function can be called as follows:

In C/C++:

```
#include <blis.h>
...
gint_t info_value = bli_info_get_info_value();
```

In Fortran:

```
integer :: info_value
integer, external :: bli_info_get_info_value
...
info_value = bli_info_get_info_value()
```

If the returned value is not zero, the value indicates the argument in the preceding BLAS call that was incorrect.

Note: Errors from an incorrect setting of the `BLIS_ARCH_TYPE` environment variable (used to override the default choice in dynamic dispatch, refer to section 4.4.1.3 for details) are handled by a separate error mechanism and will not be affected by the environment variables `BLIS_STOP_ON_ERROR` and `BLIS_PRINT_ON_ERROR`.

4.5.2 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=0x7fffffffcfef0, 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=0x7fffffffcfef0, 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)
```

4.5.3 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.

4.5.3.1 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 25.

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.

4.5.3.2 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 25.

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 $
```

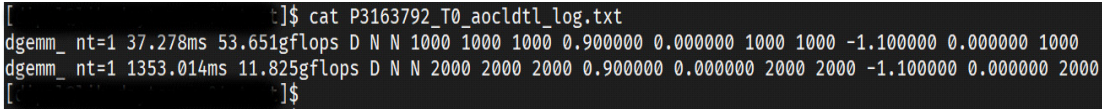
Figure 2. Sample Run with Debug Logs Enabled

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

4.5.3.3 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.

4.5.4 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:

```

dim_t AOCL_progress(
const char* const api,
const dim_t 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 10. 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:

```
dim_t AOCL_progress(
const char* const api,
const dim_t 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 15/16 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 3.15 through 3.23.3
- Microsoft Visual Studio 2019 (build 16.8.7) and 2022 (build 17.3.2 through 17.7.5)
- Microsoft Visual Studio tools (as shown in Figure 4):
 - Python development
 - Desktop development with C++: C++ Clang-Cl for v142 build tool (x64/x86)

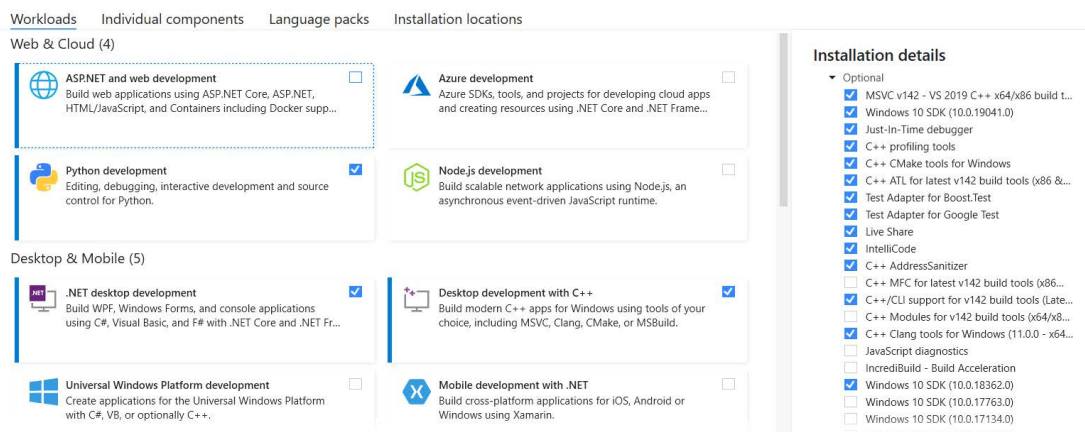


Figure 4. Microsoft Visual Studio Prerequisites

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:

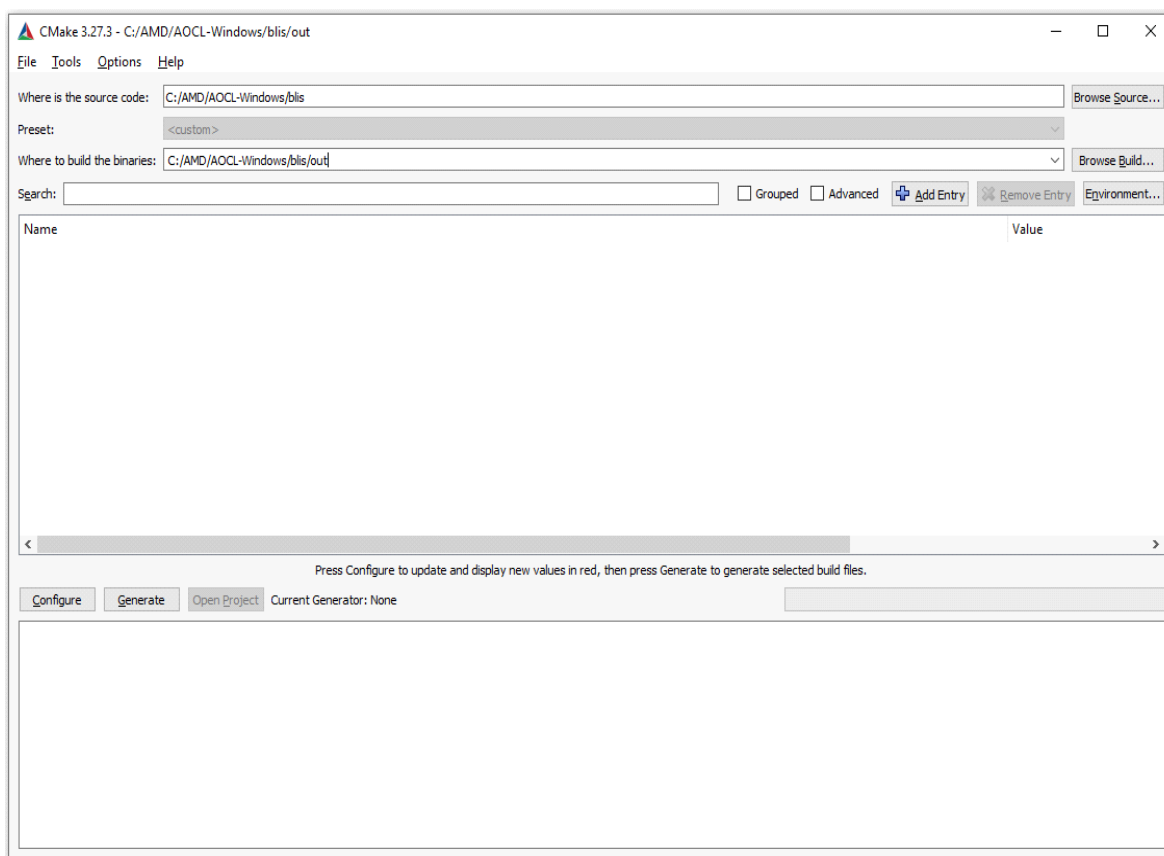


Figure 5. CMake Source and Build Folders

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 17 2022** and the compiler to **ClangCl** as shown in the following figure:

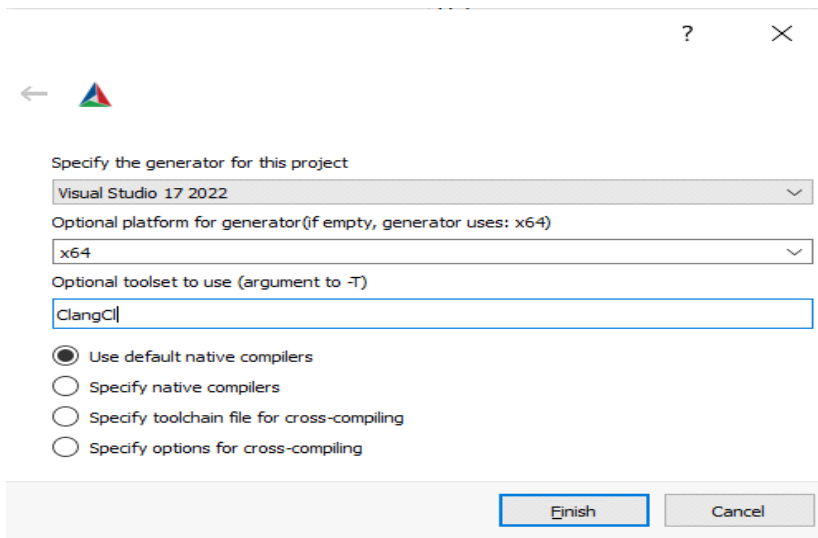


Figure 6. Set Generator and Compiler

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

Table 11. CMake Config Options

Feature	CMake Parameter
AMD CPU architecture	BLIS_CONFIG_FAMILY=zen/zen2/zen3/zen4/amdzen
Shared library	BUILD_SHARED_LIBS=ON
Static library	BUILD_SHARED_LIBS=OFF
Debug/Release build type	CMAKE_BUILD_TYPE=Debug/Release
Enable single threading (disables AOCL dynamic dispatch)	ENABLE_THREADING=no(default)
Enable multi-threading (enables AOCL dynamic dispatch with OpenMP)	ENABLE_THREADING=openmp
AOCL Dynamic (automatically selected depending on the value of ENABLE_THREADING)	ENABLE_AOCL_DYNAMIC=ON/OFF
Enable BLAS/CBLAS support	ENABLE_BLAS=ON ENABLE_CBLAS=ON
Enable 32-bit integer size in BLIS and BLAS APIs	INT_SIZE=32 and BLAS_INT_SIZE=32
Enable 64-bit integer size in BLIS and BLAS APIs	INT_SIZE=64 and BLAS_INT_SIZE=64
Absolute path to the OpenMP library, including the library name	OpenMP_libomp_LIBRARY

Table 11. CMake Config Options

Feature	CMake Parameter
All variables and their default values	BUILD_SHARED_LIBS=ON(default)/OFF ENABLE_THREADING=no(default)/openmp INT_SIZE=auto(default)/32/64 BLAS_INT_SIZE=32(default)/64 ENABLE_BLAS=ON/OFF(default) ENABLE_CBLAS=ON/OFF(default) ENABLE_MIXED_DT=ON(default)/OFF ENABLE_SUP_HANDLING=ON(default)/OFF ENABLE_AOCL_DYNAMIC=ON(default)/OFF COMPLEX_RETURN=gnu(default)/intel ENABLE_NO_UNDERSCORE_API=ON/OFF(default) ENABLE_UPPERCASE_API=ON/OFF(default) ENABLE_SYSTEM=ON(default)/OFF THREAD_PART_JRIR=slab(default)/rr ENABLE_PBA_POOLS=ON(default)/OFF ENABLE_SBA_POOLS=ON(default)/OFF ENABLE_MEM_TRACING=ON/OFF(default) ENABLE_MIXED_DT=ON(default)/OFF ENABLE_MIXED_DT_EXTRA_MEM=ON(default)/OFF ENABLE_SUP_HANDLING=ON(default)/OFF ENABLE_TRSM_PREINVERSION=ON(default)/OFF FORCE_VERSION=no(default)/<user-defined> DISABLE_BLIS_ARCH_TYPE=ON/OFF(default) RENAME_BLIS_ARCH_TYPE=BLIS_ARCH_TYPE(default)/<user-defined> RENAME_BLIS_MODEL_TYPE=BLIS_MODEL_TYPE(default)/<user-defined>

For the detailed documentation of all the options, configure CMake with `PRINT_CONFIGURE_HELP=ON`.

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

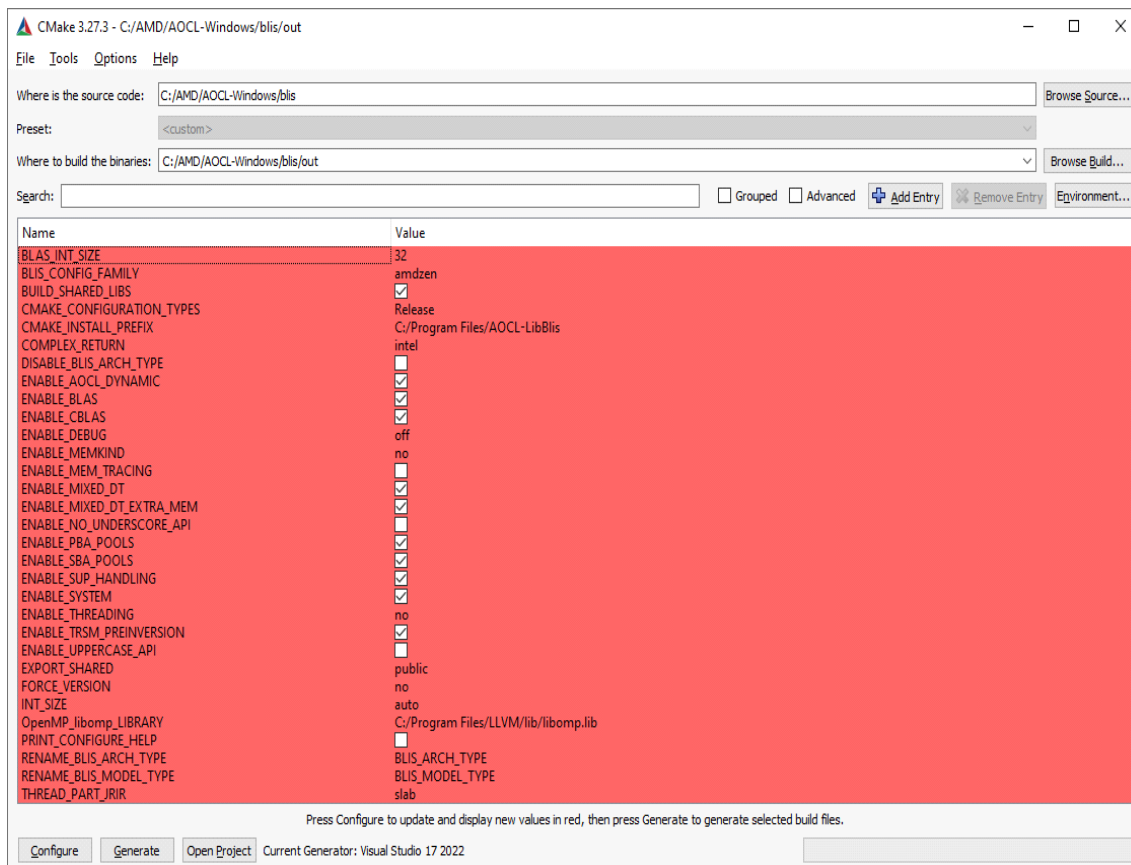


Figure 7. CMake Configure and Generate Project Settings

4.6.1.2 Building the Project in Visual Studio GUI

Complete the following steps in the Microsoft Visual Studio GUI:

- Open the project generated by CMake (build folder) in “Preparing Project with CMake GUI” on page 53.
- To generate AOCL-BLAS binaries, build the **AOCL-LibBlis** project or **libs/libblis** target.
The library files will be generated in the **out** folder based on the project settings.
For example, *blis/out/Release/AOCL-LibBlis-Win-MT.dll* or *AOCL-LibBlis-Win-MT.lib*
- To install the binaries (or to build and install them), build the **INSTALL** project under **CMakePredefinedTargets**.

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 17 2022" -DCMAKE_BUILD_TYPE=Release
-DDBLIS_CONFIG_FAMILY=amdzen -DBUILD_SHARED_LIBS=ON -DENABLE_THREADING=openmp
-DCOMPLEX_RETURN=intel -DOpenMP_libomp_LIBRARY="C:\Program
Files\LLVM\lib\libomp.lib" -TClangCL
```

You can refer Table 11 and update the parameter options in the command according to the project requirements or run the following command for a detailed description of the available options:

```
cmake -S .. -B . -G "Visual Studio 17 2022" -DPRINT_CONFIGURE_HELP=ON
```

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
```

For building the library using multiple threads, run the following command:

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

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

4.6.3 Packaging AOCL -BLAS

During CMake configuration, AOCL-BLAS package will be created in the specified path using:

```
-DCMAKE_INSTALL_PREFIX=<blas_install_path>
```

The AOCL-BLAS package contains *lib* and *include* folders.

4.6.4 Building and Running the Test Suite

To build and run the test suite, execute the following command:

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

To build the test suite using multiple threads, execute the following command:

```
cmake --build . --config Release --target checkblis -j
```


The same header can be used for both static and shared libraries on Windows. To access DLL's public data symbols and objects, you can define `BLIS_EXPORT=__declspec(dllimport)` to import those symbols explicitly. Importing is not required for:

- The AOCL-BLAS and CBLAS interface users
- Most of the cases where BLIS interface is used

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 'A' is unsigned 8-bit (u8) and matrix 'B' signed 8-bit (s8), accumulation matrix 'C' 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, clipping 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 repeated access to memory 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 APIs and Post-ops in aocl_gemm

4.7.4.1 Architecture Features and APIs

Table 12. Required Architecture Features and APIs

Architecture Features Required	API
AVX512-VNNI	aocl_gemm_u8s8s32os32
	aocl_gemm_u8s8s32os8
	aocl_gemm_s8s8s32os32
	aocl_gemm_s8s8s32os8
AVX2	aocl_gemm_u8s8s16os16
	aocl_gemm_u8s8s16os8
	aocl_gemm_u8s8s16ou8
	aocl_gemm_s8s8s16os16
	aocl_gemm_s8s8s16os8
AVX512-BF16	aocl_gemm_bf16bf16f32of32
	aocl_gemm_bf16bf16f32obf16
AVX512	aocl_gemm_f32f32f32of32

4.7.4.2 Utility APIs in aocl_gemm Add-on

Table 13. GEMM API Supported Post-ops

Post-op	Description
Add bias	Adds bias to the GEMM output before storing into C, where the bias data is passed by the user using the post-op interface.
ReLU	Performs ReLU operation on GEMM output. $f(x) = 0$, when $x \leq 0$ and $f(x) = x$ when $x > 0$.
PReLU	Performs Parametric ReLU operation on GEMM output based on scale given by the user. $f(x) = x$, when $x > 0$ and $f(x) = \text{scale} * x$ when $x \leq 0$.
GeLU-Tanh	Perform Tanh based GeLU on GEMM output. $\text{GeLU_Tanh}(x) = 0.5 * x * (1 + \tanh(0.797884 * (x + (0.044715 * x^3))))$
GeLU-ERF	Perform Erf based GeLU on GEMM output. $\text{GeLU_Erf}(x) = 0.5 * x * (1 + \text{erf}(x * 0.707107))$
Scale	Perform Scale operation on GEMM output based on the scale provided by the user.
Clip	Perform clip operation on GEMM output based on minimum and maximum values given by the user.

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 14. Utility APIs in aocl_gemm Add-on

API	Description
<code>aocl_get_reorder_buff_size_XXX XXXXXX()</code>	Returns buffer size required to reorder an input matrix, where XXXXXXXX corresponds to each of the data type combinations specified in Table 15 . For example, u8s8s32os32.
<code>aocl_reorder_XXXXXXX ()</code>	Reorders the given input and writes into output buffer.
<code>aocl_gelu_tanh_f32()</code>	Performs tanh operation on each element of the given input buffer and writes in the output buffer.
<code>aocl_gelu_erf_f32()</code>	Performs tanh operation on each element of the given input buffer and writes in the output buffer.
<code>aocl_softmax_f32()</code>	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 on Linux:

- 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 15](#). The APIs returns without doing anything when those features are not available.
- Transpose support for A and B is currently available only for bf16 APIs.

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.x -I/aocl-blis_install_directory/include/amdzen/  
-L/aocl-blis_install_directory/lib/amdzen/ -lblis-mt -lm
```

Note: Export blis lib path to LD_LIBRARY_PATH before running the executable

```
*/  
  
// Add Update the LD_LIBRARY_PATH with blis library path and run ./test_lpgem.x  
#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;
```



```
err_t err = BLIS_SUCCESS;
uint8_t *a = (uint8_t *)bli_malloc_user(sizeof(uint8_t) * m * k, &err);
int8_t *b = (int8_t *)bli_malloc_user(sizeof(int8_t) * n * k, &err);
int32_t *c = (int32_t *)bli_malloc_user(sizeof(int32_t) * m * n, &err);
if (err != BLIS_SUCCESS)
{
    printf("Memory allocation Failed\n");
    goto bailout;
}

// 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 for all API's.
char transb = 'n';
char reordera = 'n';
char reorderb = 'n';

aocl_gemm_u8s8s32os32(
    storage, transa, transb,
    m, n, k,
    alpha,
    a, lda, reordera,
    b, ldb, reorderb,
    beta,
    c, ldc,
    NULL);

bailout:
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_postops.c -o test_lpgemm_postops.x -I/aocl-blis_install_directory/include/
amdzen
-L/aocl-blis_install_directory/lib/ -lBLIS-mt -lm
Note: Export blis lib path to LD_LIBRARY_PATH before running the executable
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "blis.h"
// Example program to demonstrate usage of LPGEMM downscale API with post-ops.
// 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;
    err_t err = BLIS_SUCCESS;
    uint8_t *a = (uint8_t *)bli_malloc_user(sizeof(uint8_t) * m * k, &err);
    int8_t *b = (int8_t *)bli_malloc_user(sizeof(int8_t) * n * k, &err);
    int8_t *c = (int8_t *)bli_malloc_user(sizeof(int8_t) * m * n, &err);
    if (err != BLIS_SUCCESS)
        goto bailout;
    // 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 = 'n';
    // Initialize post-ops struct.
    aocl_post_op *post_ops = NULL;
    post_ops = (aocl_post_op *)bli_malloc_user(sizeof(aocl_post_op), &err);
    if (err != BLIS_SUCCESS)
        goto bailout;
    // 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 *)
        bli_malloc_user(max_post_ops_seq_length * sizeof(AOCL_POST_OP_TYPE),
            &err);
    if (err != BLIS_SUCCESS)
        goto bailout;

```



```

// Bias
post_ops->seq_vector[0] = BIAS;
// Need to output accumulation (int32_t) type for bias.
post_ops->bias.bias = bli_malloc_user(n * sizeof(int32_t), &err);
if (err != BLIS_SUCCESS)
    goto bailout;
// 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 = bli_malloc_user(2 * sizeof(aocl_post_op_eltwise), &err);
if (err != BLIS_SUCCESS)
    goto bailout;
// 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 = bli_malloc_user(sizeof(int32_t), &err);
if (err != BLIS_SUCCESS)
    goto bailout;
// Max bound is represented by beta.
(post_ops->eltwise + 1)->algo.beta = bli_malloc_user(sizeof(int32_t), &err);
if (err != BLIS_SUCCESS)
    goto bailout;
// 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.buff = NULL;
post_ops->sum.zero_point = bli_malloc_user(n * sizeof(float), &err);
if (err != BLIS_SUCCESS)
    goto bailout;
post_ops->sum.scale_factor = bli_malloc_user(n * sizeof(float), &err);
// Add function to fill downscale array here.
post_ops->seq_length = 4;
if (err != BLIS_SUCCESS)
    goto bailout;

```



```

aocl_gemm_u8s8s32os8(
    storage, transa, transb,
    m, n, k,
    alpha,
    a, lda, reordera,
    b, ldb, reorderb,
    beta,
    c, ldc,
    post_ops);

bailout:
    if (post_ops->sum.scale_factor != NULL)
    {
        bli_free_user(post_ops->sum.scale_factor);
    }
    if ((post_ops->eltwise + 1)->algo.alpha != NULL)
    {
        bli_free_user((post_ops->eltwise + 1)->algo.alpha);
    }
    if ((post_ops->eltwise + 1)->algo.beta != NULL)
    {
        bli_free_user((post_ops->eltwise + 1)->algo.beta);
    }
    if (post_ops->eltwise != NULL)
    {
        bli_free_user(post_ops->eltwise);
    }
    if (post_ops->bias.bias != NULL)
    {
        bli_free_user(post_ops->bias.bias);
    }
    if (post_ops->seq_vector != NULL)
    {
        bli_free_user(post_ops->seq_vector);
    }
    if (post_ops != NULL)
    {
        bli_free_user(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: *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-Utils library

Build Steps

AOCL-LAPACK supports compiling the library using CMake build system in addition to 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.

Note: AOCL-LAPACK depends on the AOCL Utilities library (AOCL-Utills) for certain functions including CPU architecture detection at runtime. The default build of AOCL-LAPACK requires the path to the AOCL-Utills header files to be set. The applications using AOCL-LAPACK must link with the AOCL-Utills library explicitly.

Method 1: Using CMake

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

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

2. Run the following command to configure the project:

Set header file path of the AOCL-Utills library using the LIBAOCLUTILS_INCLUDE_PATH option:

- With GCC (default):

Using 32-bit Integer (LP64)

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

Using 64-bit Integer (ILP64)

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

- 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> -
DLIBAOCLUTILS_INCLUDE_PATH=<path_to_libaoclutils_header_files>
```

Using 64-bit Integer (ILP64)

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

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

Method 2: Using Configure/Makefile

1. Set the header file path of AOCL-Utills in CFLAGS environment variable:

```
$ export CFLAGS="-I<path to libaoclutils include directory>"
```
2. 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>
```

3. 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*.

Linking with AOCL-BLAS

AOCL-LAPACK can be linked with any Netlib BLAS compliant library when compiled with standard CMake options above. However, AOCL-LAPACK provides an option explicitly to link explicitly with AOCL-BLAS library at compile time. This option enables invoking lower level AOCL-BLAS APIs directly and that could result in better performance for certain APIs on AMD "Zen" CPUs. To force AOCL-LAPACK to use AOCL-BLAS library, provide the option `ENABLE_AOCL_BLAS` in the CMake configuration:

```
$ cmake -DENABLE_AMD_AOCC_FLAGS=ON -DENABLE_AOCL_BLAS=ON ...
```

Provide path of the AOCL-BLAS library using one of the following methods:

- Set "AOCL_ROOT" environment variable to the root path where AOCL-BLAS library(\$AOCL_ROOT/lib) and header files(\$AOCL_ROOT/include) are located:

```
$ export AOCL_ROOT=<path to AOCL-BLAS>
```
- Specify root path of the AOCL-BLAS library through the CMake option "AOCL_ROOT":

```
$ cmake -DENABLE_AMD_AOCC_FLAGS=ON -DENABLE_AOCL_BLAS=ON -DAOCL_ROOT=<path to AOCL-BLAS> ...
```

The path specified in `AOCL_ROOT` must have the directories "include" and "lib" containing the necessary header files and AOCL-BLAS binary respectively.

Auto-linking AOCL-Utills Library

The option to merge the AOCL-Utills library with the AOCL-LAPACK library is deprecated and not recommended in most scenarios. This is provided as backward support of previous release.

Auto-linking of AOCL-Utills can be done using "ENABLE_EMBED_AOCLUTILS" option for both CMake and autoconfigure tools build mode. With this option, AOCL-LAPACK automatically links with libaoclutils library by downloading the source of libaoclutils from AMD GitHub, compiling it and linking/merging with AOCL-LAPACK library. A sample command is as follows:

CMake Build:

```
$ cmake ../ -DENABLE_AMD_FLAGS=ON -DENABLE_EMBED_AOCLUTILS=ON
```

Autoconfigure:

```
$ configure --enable-amd-flags
$ make ENABLE_EMBED_AOCLUTILS=1 -j
```

With embed AOCL-Utills build, if an external path is provided for the libaoclutils binary and header files through separate flags, 'LIBAOCLUTILS_LIBRARY_PATH' and

'LIBAOCLUTILS_INCLUDE_PATH' respectively, the specified library is used instead of downloading from GitHub. A sample command is as follows:

CMake Build:

```
$ cmake ../ -DENABLE_AMD_FLAGS=ON -DENABLE_EMBED_AOCLUTILS=ON DLIBAOCLUTILS_LIBRARY_PATH=<path/to/libaoclutils/library> -DLIBAOCLUTILS_INCLUDE_PATH=<path/to/libaoclutils/header/files>
```

Autoconfigure:

```
$ make ENABLE_EMBED_AOCLUTILS=1 LIBAOCLUTILS_LIBRARY_PATH=<path/to/libaoclutils/library> LIBAOCLUTILS_INCLUDE_PATH=<path/to/libaoclutils/header/files> -j
```

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 AVX2, FMA, or AVX512 depending on the ISA supported on the target CPU, you can use the following steps:

Using CMake:

Set the flag `LF_ISA_CONFIG` to the desired ISA support. The available options are Auto, AVX2 (default), AVX512, and None. The command to use this is as follows:

```
$ cmake .. -DLF_ISA_CONFIG=AVX512 -DENABLE_AMD_FLAGS=ON
```

Using Configure/Makefile:

Set the configure option `--enable-optimizations` to the desired optimization flags and 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 19*.

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 critical set of AOCL-LAPACK APIs and new test cases are being added on an -ongoing basis. 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, AOCL-BLAS, and AOCL-Utills libraries while building the application.

AOCL-Utills library has libstdc++ library dependency. As AOCL-LAPACK is dependent on AOCL-Utills, applications must link with libstdc++(-lstdc++) as well.

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> -DAOCLUTILS_LIBRARY_PATH=<path to aoclutils library>
```

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 -DAOCLUTILS_LIBRARY_PATH=/home/usr/aoclutils-
install/lib
```

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 52*. Also, AOCL-LAPACK has dependency on AOCL-Utils library.

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 15. AOCL-LAPACK Config Options

Feature	CMake Parameter(s)
Shared library	BUILD_SHARED_LIBS=ON
Static library	BUILD_SHARED_LIBS=OFF
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_FLAME_MALLOC ENABLE_BUILTIN_LAPACK2FLAME ENABLE_EXT_LAPACK_INTERFACE ENABLE_INTERNAL_ERROR_CHECKING ENABLE_NON_CRITICAL_CODE ENABLE_PORTABLE_TIMER INCLUDE_LAPACKE

Table 15. AOCL-LAPACK Config Options

Feature	CMake Parameter(s)
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 <i>Note: You can set it to OFF to build a single-threaded AOCL-LAPACK library.</i>
On Windows, setting ENABLE_AMD_FLAGS flag internally enables: <ul style="list-style-type: none"> • ENABLE_BLAS_EXT_GEMMT • ENABLE_AMD_OPT • ENABLE_BUILTIN_LAPACK2FLAME • ENABLE_EXT_LAPACK_INTERFACE • ENABLE_F2C_DOTC • ENABLE_VOID_RETURN_COMPLEX_FUNCTION • ENABLE_MULTITHREADING 	ENABLE_AMD_FLAGS=ON
Set external libaoclutils library path	LIBAOCLUTILS_LIBRARY_PATH=<path to libaoclutils library>
Set external libaoclutils header path	LIBAOCLUTILS_INCLUDE_PATH=<path to libaoclutils header files path>
Enable main test suite	BUILD_TEST=ON (ensure that BUILD_LEGACY_TEST is not set)
Enable legacy test suite	BUILD_LEGACY_TEST=ON (ensure that BUILD_TEST is not set)
Set BLAS library header path	BLAS_HEADER_PATH=<path to BLAS header file>

Table 15. AOCL-LAPACK Config Options

Feature	CMake Parameter(s)
Enable Netlib test suite	BUILD_NETLIB_TEST=ON
Set the instruction set architecture (ISA) to compile with Valid options: Auto, AVX2, AVX512, and None	-- LF_ISA_CONFIG=[options]
Enable embedding the AOCL-Utils library in AOCL-LAPACK	-- ENABLE_EMBED_AOCLUTILS=OFF
Enable tight-coupling with the AOCL-BLAS library to use AOCL-BLAS internal routines	-- ENABLE_AOCL_BLAS=OFF <i>Note: Enables invoking lower level AOCL-BLAS APIs directly and that could result in better performance for certain APIs on AMD "Zen" CPUs.</i>
Set the AOCL-BLAS installation path	-- AOCL_ROOT=<path to AOCL-BLAS install directory having include and lib folders>

5. Provide the path to the BLAS and AOCL-Utils libraries. 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:

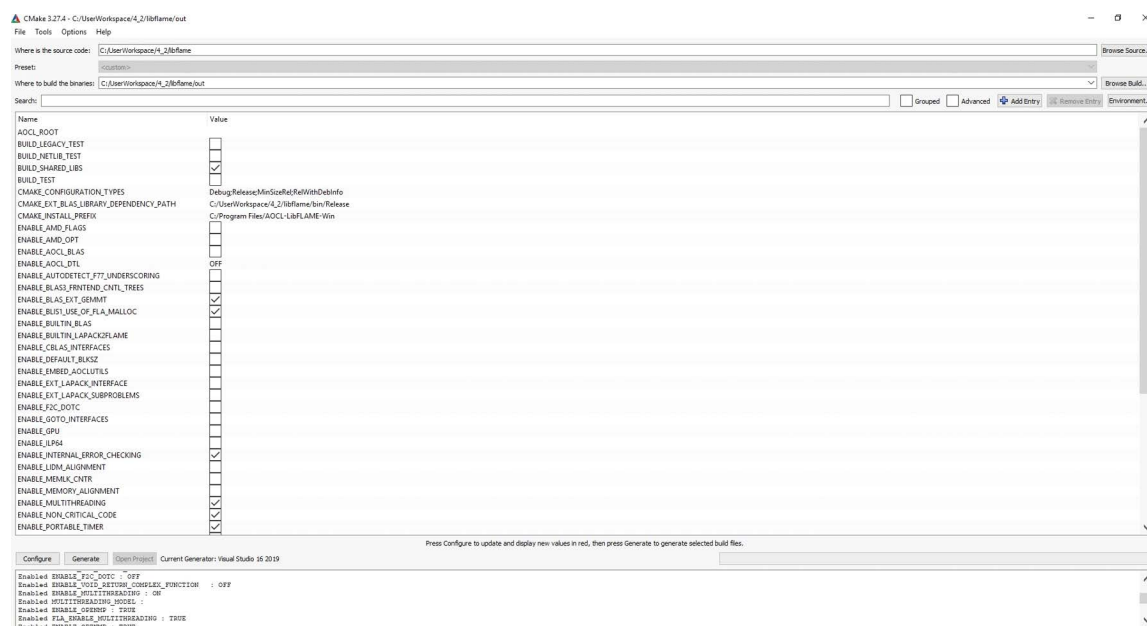


Figure 8. AOCL-LAPACK CMake Configurations

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 75.
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>" -
DLIBAOCLUTILS_LIBRARY_PATH="<path to AOCL-Utills library including library>" -
DLIBAOCLUTILS_INCLUDE_PATH="<path to AOCL-Utills header files>" -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"
```

Note: Add `-DENABLE_MULTITHREADING=OFF` to build a single-threaded AOCL-LAPACK library.

You can refer to [Table 15](#) 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 15, enable "BUILD_TEST" to build a new main test suite of AOCL-LAPACK. To build a legacy test suite, set "BUILD_LEGACY_TEST".

Note: On Windows, 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 16. 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 in the current threads team

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);
```

Note: *In the case of single-threaded AOCL-LAPACK (--enable-multithreading=none or ENABLE_MULTITHREADING=OFF), the values of "current_thread" and "total_threads" are set to 0 and 1 respectively. As a result, the callback function cannot be used to monitor the thread ID and thread count of the application.*

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 164.

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 17. 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 17. 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:

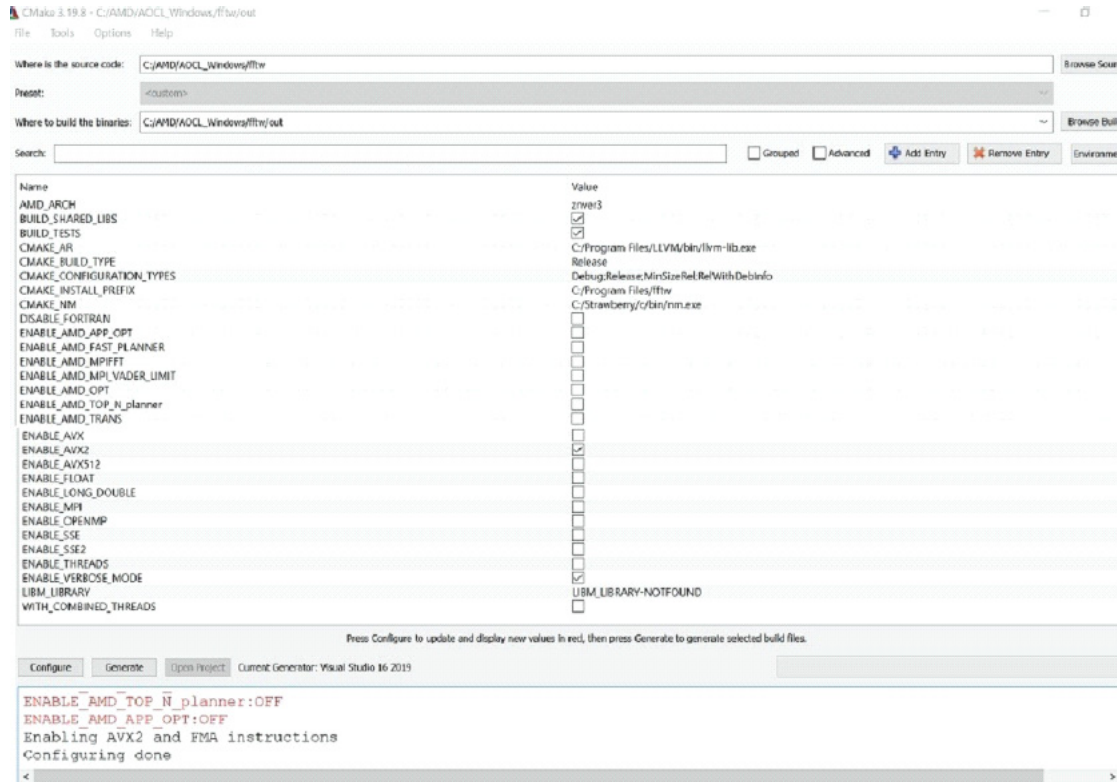


Figure 9. AOCL-FFTW CMake Config Options

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 17 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 19.

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 164.

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 164.

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 164.

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 elementary 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, in which a small amount of the accuracy has been traded for greater performance.

7.1 Library Contents

7.1.1 Scalar Functions

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 Function
 - erff and erf
- Power
 - powf, pow, cbrtf, cbrt, sqrtf, sqrt, hypotf, and hypot
- 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
 - fabsf and fabs
 - copysignf, copysign, nanf, nan, finitf, and finite
 - nextafterf, nextafter, nexttowardf, and nexttoward
- Maximum, Minimum, and Positive Difference
 - fmaxf, fmax, fminf, fmin, fdimf, and fdim

Also, there are a small number of complex scalar functions: cpowf, cpow, clogf, clog, cexpf, and cexp.

7.1.2 Fast Scalar and Vector Variants

Faster but less accurate versions of some of the scalar functions are available in the library *libalmfast.so*. It contains fast versions of acosf, acos, asinf, asin, atanf, atan, cosf, cos, erff, erf, expf, exp, logf, log, powf, pow, sinf, sin, tanf, and tan. 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 4.2 user guide.

AOCL-LibM includes vector variants for many of the core math functions as listed later in this section. A few caveats on both the fast scalar versions and the vector variants are as follows:

- These routines 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.
- These variants do not set the IEEE error codes and hence, the user code must not rely on them doing so.

The vector variants can be enabled by using the AOCC compiler with the `-ffast-math -fvecLib=AMDLIBM` flags. 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 for the vector functions:

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

where,

- `v` – vector
- `r` – real
- `<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; or ‘a’ if this is a vector array function
- `<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_cosf, vrs8_cosf, vrs4_sinf, and vrs8_sinf
 - vrsa_cosf, vrsa_sinf, and vrsa_sincosf
 - 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_cosf, vrs16_sinf, and vrs16_tanf
 - vrd8_cos, vrd8_sin, vrd8_tan, and vrd8_sincos
- Inverse Trigonometric
 - vrs4_acosf, vrs4_asinf, and vrs8_asinf
 - vrs4_atanf, vrs8_atanf, and vrd2_atan
 - vrs16_atanf, vrs16_asinf, and vrs16_acosf
 - 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 Function
 - 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
 - vrsa_fmaxf, vrsa_fmaxfi, vrda_fmax, and vrda_fmaxi
 - vrsa_fminf, vrsa_fminfi, vrda_fmin, and vrda_fmini

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

It is recommend to use a GCC version of 9.2 or later as 9.2 is the version at which AMD “Zen2” compiler optimizations were introduced.

AMD “Zen3” compiler optimizations were added at GCC 10.3 and AMD “Zen4” at 12.3.

- Clang 12.0.0 (AOCC 3.0) through Clang 17.0.0 (AOCC 4.2)
- Virtualenv with Python 3.6 or later
- SCons version 3.1.1 or later
- libstdc++ (required for AOCL-Utils)

The minimum and maximum permitted versions of GCC and Clang are set in the file *scripts/site_scons/alm/check.py*. You can edit it to allow the use of other compiler versions.

Refer to [Chapter 3](#) to install the AOCL-Utills 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 the branch *aocl-4.2*:

```
cd aocl-libm-ose
git checkout aocl-4.2
```

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. By default, the libraries (static and dynamic) will be compiled and generated in the following location:

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

If a *debug mode build* has been selected, the libraries (static and dynamic) will instead be compiled and generated in the following location:

aocl-libm-ose/build/aocl-debug/src

If the *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-Utills)

Refer to [Chapter 3](#) to install the AOCL-Utills 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 the 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
```

By default, 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/

If a *debug mode build* has been selected, the libraries will instead be compiled and generated in the following location:

aocl-libm-ose/build/aocl-debug/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, *-la1m* 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> -la1m -lm (cc can be ‘gcc’ or
  ‘clang’).
$ ./myprogram
```

You can access the vector calls by using the AOCC compiler with the flags *-ffast-math -fveclib=AMDLIBM*.

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.

Following table lists the CMake flags supported by AOCL-ScaLAPACK:

Table 18. 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> <i>Note: If linked with AOCL-LAPACK, include the AOCL-Utils library path too.</i>
Integer bit length: • ON => 64-bit integer length • OFF => 32-bit integer length	ENABLE_ILP64
Flags disabled by default	USE_OPTIMIZED_LAPACK_BLAS
Set OpenMP library path	OpenMP_libomp_LIBRARY=<path to OpenMP library>
Enable Trace and Log feature	ENABLE_DTL=OFF
Configurable flag for Windows builds	CDEFS = NoChange (Fortran to C interface API's called with small case) CDEFS = UpCase (Fortran to C interface API's called with upper case) CDEFS = Add_ (Fortran to C interface API's called with small case and underscore) DCDEFS=NoChange

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-Utils
- 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 aocl-scalapack
```

3. CMake as follows:

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

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

- b. Export PATH and LD_LIBRARY_PATH to the *lib* and *bin* folders of the MPI installation respectively:

```
export PATH=<MPI installation folder>/bin:$PATH  
export LD_LIBRARY_PATH=<MPI installation folder>/lib:$LD_LIBRARY_PATH
```

- c. Run `cmake` command based on the compiler and the type of library generation required.

Note: *AOCL-LAPACK is dependent on the AOCL-Utils library, which in turn depends on libstdc++. Hence, you must link with AOCL-Utils and libstdc++(-lstdc++) along with the AOCL-LAPACK library while specifying the path for LAPACK_LIBRARIES in the CMake flags.*

Table 19. Compiler and Type of Library

Compiler	Library Type	Threading	Command
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 <path to AOCL-Utills library>/libaoclutils.a " -DCMAKE_C_COMPILER=mpicc -DCMAKE_Fortran_COMPILER=mpif90 -DUSE_OPTIMIZED_LAPACK_BLAS=OFF [-DDENABLE_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 <path to AOCL-Utills library>/libaoclutils.a " -DCMAKE_C_COMPILER=mpicc -DCMAKE_Fortran_COMPILER=mpif90 -DUSE_OPTIMIZED_LAPACK_BLAS=OFF [-DDENABLE_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-LAPACK library>/libflame.so <path to AOCL-Utills library>/libaoclutils.so " -DCMAKE_C_COMPILER=mpicc -DCMAKE_Fortran_COMPILER=mpif90 -DUSE_OPTIMIZED_LAPACK_BLAS=OFF [-DDENABLE_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-LAPACK library>/libflame.so <path to AOCL-Utills library>/libaoclutils.so " -DCMAKE_C_COMPILER=mpicc -DCMAKE_Fortran_COMPILER=mpif90 -DUSE_OPTIMIZED_LAPACK_BLAS=OFF [-DDENABLE_ILP64=ON]

Table 19. Compiler and Type of Library

Compiler	Library Type	Threading	Command
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 <path to AOCL-Utills library>/libaoclutils.a" -DCMAKE_C_COMPILER=mpicc -DCMAKE_Fortran_COMPILER=mpif90 -DUSE_OPTIMIZED_LAPACK_BLAS=OFF [-DDENABLE_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 <path to AOCL-Utills library>/libaoclutils.a" -DCMAKE_C_COMPILER=mpicc -DCMAKE_Fortran_COMPILER=mpif90 -DUSE_OPTIMIZED_LAPACK_BLAS=OFF [-DDENABLE_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-LAPACK library>/libflame.so <path to AOCL-Utills library>/libaoclutils.so" -DCMAKE_C_COMPILER=mpicc -DCMAKE_Fortran_COMPILER=mpif90 -DUSE_OPTIMIZED_LAPACK_BLAS=OFF [-DDENABLE_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-LAPACK library>/libflame.so <path to AOCL-Utills library>/libaoclutils.so" -DCMAKE_C_COMPILER=mpicc -DCMAKE_Fortran_COMPILER=mpif90 -DUSE_OPTIMIZED_LAPACK_BLAS=OFF [-DDENABLE_ILP64=ON]

On Linux, the inbuilt communications sub-module of AOCL-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 <path to AOCL-Utills library>/libaoclutils.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 <path to AOCL-Utils library>/libaoclutils.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 19.

8.2 Usage

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

```
$ cd aocl-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-Utills 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
- Intel MPI 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. Select the available and recommended options as follows:

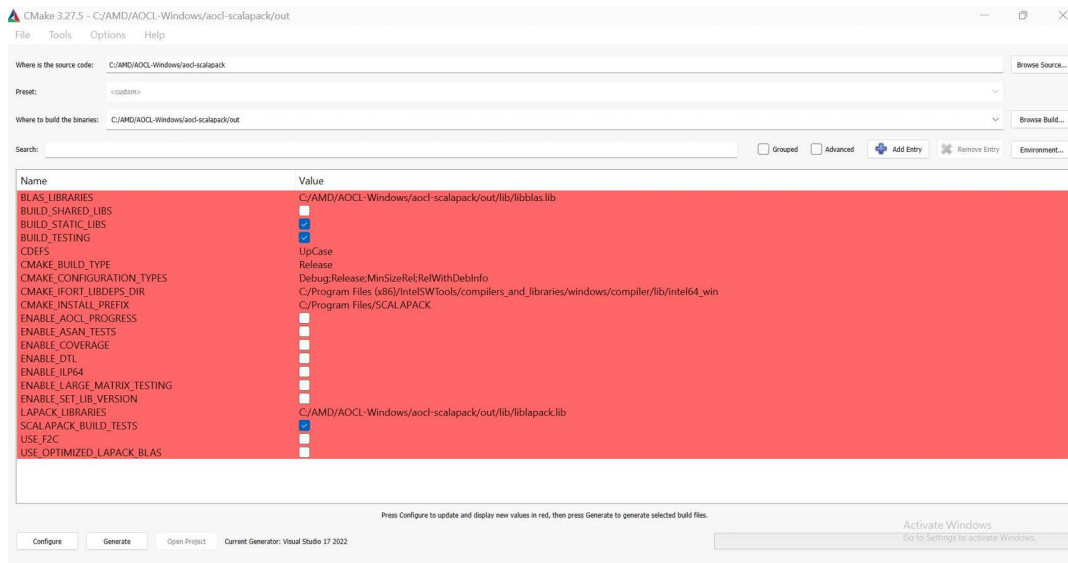


Figure 10. AOCL-ScaLAPACK CMake Options

5. 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 103*.
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 -DCDEFs=UpCase /NoChange/Add_
-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;<path to AOCL-
Utils library>/libaoclutils.lib"
```

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 20. 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 21. Additional Features

Feature	Description	Environment Variable	OS Support
Trace	Enable function call trace.	AOCL_SL_TRACE	Linux, Windows
Log	Enable logging of input argument values.	AOCL_SL_LOG	Linux, Windows

Table 21. Additional Features

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
---------------	--	------------------	----------------

Note: To use the Trace and Log feature, ensure that AOCL-ScaLAPACK is built with the flag `ENABLE_DTL=ON`.

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.

For more information, refer to the RNG documentation <AOCL team to provide hyperlink>.

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 19*.

To install the AOCL-RNG binary for Windows, refer to *"Using Windows Packages" on page 23*. *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 93*.
- To install AOCL-BLAS binary for linux and Windows, refer to *"Using Pre-built Binaries" on page 27*.
- 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.

For more information, refer to the SecureRNG documentation <AOCL team to provide hyperlink>.

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 19.

To install the AOCL-SecureRNG binary for Windows, refer to “*Using Windows Packages*” on page 23. *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 (Sparse BLAS). It is designed to be used with C, C++, and compatible languages.

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

- **Sparse Level 1** functions perform vector operations, such as dot product, vector additions on sparse vectors, gather, scatter, and other similar operations.
- **Sparse Level 2** functions describe the operations between a matrix in sparse format and a vector in dense format, including matrix-vector product (SpMV), triangular solve (TRSV), and so on.
- **Sparse Level 3** functions describe the operations between a matrix in sparse format and a matrix in dense/sparse format. The operations comprise of matrix additions (SpADD), matrix-matrix product (SpMM, Sp2M), and triangular solver with multiple right-hand sides (TRSM).
- Iterative sparse solvers based on Krylov subspace methods (CGM, GMRES) and pre-conditioners (such as SymGS and ILU0).
- Sparse format conversion functions for translating matrices in a variety of sparse storage formats.
- Auxiliary functions to allow basic operations, including create, copy, destroy, and modify matrix handles and descriptors.

Additional Highlights

- Supported data types: single, double, and the complex variants
- 0-based and 1-based indexing of sparse formats

Hint & Optimize framework to accelerate the supported functions by a prior matrix analysis based on the users' hints of expected operations. **Multi-thread Support**

Note: 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 function with multi-thread support includes SpMV variants and TRSM. *Currently, only single-threaded builds are supported on Windows; multi-threaded support will be added in future release.*

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

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

11.1.1 Building AOCL-Sparse from Source on Linux

The following prerequisites must be met:

- Git
- CMake versions 3.11 through 3.27
- Boost library versions 1.65 through 1.83

Note: *This is needed only for benchmarks (BUILD_CLIENTS_BENCHMARKS=ON). For more information, refer to [Table 30](#).*

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

1. Install AOCL-BLAS, AOCL-LAPACK, and AOCL-Utills.
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 AOCL-BLAS, AOCL-LAPACK, and AOCL-Utills libraries in the same path, you can use the following variables during the CMake configuration to point to the location of the dependent libraries and headers:

- AOCL_BLIS_LIB
 - AOCL_LIBFLAME
 - AOCL_UTILS_LIB
 - AOCL_BLIS_INCLUDE_DIR
 - AOCL_LIBFLAME_INCLUDE_DIR
 - AOCL_UTILS_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. Configure the project using the following tables:

Table 22. Compiler and Library Type

Compiler	Library Type	ILP 64 Support	Command
G++ (Default)	Static	OFF (Default)	<code>cmake -S . -B out_sparse -DBUILD_SHARED_LIBS=OFF</code>
		ON	<code>cmake -S . -B out_sparse -DBUILD_SHARED_LIBS=OFF -DBUILD_ILP64=ON</code>
	Shared (Default)	OFF (Default)	<code>cmake -S . -B out_sparse</code>
		ON	<code>cmake -S . -B out_sparse -DBUILD_ILP64=ON</code>
AOCC	Static	OFF (Default)	<code>cmake -S . -B out_sparse -DCMAKE_CXX_COMPILER=clang++ -DBUILD_SHARED_LIBS=OFF</code>
		ON	<code>cmake -S . -B out_sparse -DCMAKE_CXX_COMPILER=clang++ -DBUILD_SHARED_LIBS=OFF -DBUILD_ILP64=ON</code>
	Shared (Default)	OFF (Default)	<code>cmake -S . -B out_sparse -DCMAKE_CXX_COMPILER=clang++</code>
		ON	<code>cmake -S . -B out_sparse -DCMAKE_CXX_COMPILER=clang++ -DBUILD_ILP64=ON</code>

Table 23. AOCL-Sparse - CMake Build Options

Build Option	Feature
CMAKE_INSTALL_PREFIX	Use <code>-DCMAKE_INSTALL_PREFIX=<path></code> to choose the custom path. The default install path is <code>/opt/aoclsparse/</code>
CMAKE_BUILD_TYPE	<ul style="list-style-type: none"> • Release => Release Library (Default) • Debug => Debug Library
CMAKE_CXX_COMPILER	Use <code>-DCMAKE_CXX_COMPILER=clang++</code> for AOCC builds
BUILD_SHARED_LIBS	<ul style="list-style-type: none"> • OFF => Build Static Library • ON => Build Dynamic/Shared library (Default)
BUILD_ILP64	Integer length: <ul style="list-style-type: none"> • OFF => 32-bit integer length (Default) • ON => 64-bit integer length
SUPPORT_OMP	Multi-threading using OpenMP: <ul style="list-style-type: none"> • OFF => Disable OpenMP • ON => Enable OpenMP (Default)

Table 23. AOCL-Sparse - CMake Build Options

Build Option	Feature
USE_AVX512	<ul style="list-style-type: none"> • OFF => Dynamically selects kernels (AVX2 and AVX512) for SpMV (Default) • ON => Enables AVX512 kernels for SpMV and TRSV
BUILD_CLIENTS_BENCHMARKS	<ul style="list-style-type: none"> • OFF => Disable building benchmarks (Default) • ON => Build client benchmarking (requires Boost library)
BUILD_CLIENTS_SAMPLES	<ul style="list-style-type: none"> • OFF => Disable building sparse API examples • ON => Enable building sparse examples for SPMV, CSR2M, DTRSV, CG, and GMRES (Default)
BUILD_UNIT_TESTS	<ul style="list-style-type: none"> • OFF => Unit tests are not built • ON => 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.
BUILD_DOCS	<ul style="list-style-type: none"> • ON => Build PDF and HTML documentation, this adds a new target "docs" (requires Linux and modern LaTeX distribution) • OFF => Does not activate the docs target (Default)

6. Build the AOCL-Sparse library:

```
cmake --build out_sparse --config <build_type> --parallel <nproc>
```

where,

build_type can be Release/Debug as configured during configuration.

nproc is the number of cores for the build to run in parallel.

7. Install AOCL-Sparse to the directory */opt/aoclsparse* or a custom path given by CMAKE_INSTALL_PREFIX:

```
cmake --build out_sparse --target install
```

Note: If *AOCL_ROOT* contains the shared and static libraries in the same directory for any dependent library, Sparse will link to the shared library irrespective of *BUILD_SHARED_LIBS*.

11.1.2 Building AOCL-Sparse from Source on Windows

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 [4.6](#).

11.1.2.1 Using CMake and Visual Studio GUI

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

1. Install AOCL-BLAS, AOCL-LAPACK, and AOCL-Utils.
2. Define the environment variable AOCL_ROOT to point to the AOCL Libs installation:

```
set "AOCL_ROOT=C:\Program Files\AMD\AOCL-Windows"
```

For the cases where AOCL_ROOT cannot be exported by placing AOCL-BLAS, AOCL-LAPACK, and AOCL-Utils libraries in the same path, define the following variables during CMake configuration in cmake-gui to point to the corresponding libraries and headers:

- AOCL_BLIS_LIB
- AOCL_LIBFLAME
- AOCL_UTILS_LIB
- AOCL_BLIS_INCLUDE_DIR
- AOCL_LIBFLAME_INCLUDE_DIR
- AOCL_UTILS_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**:

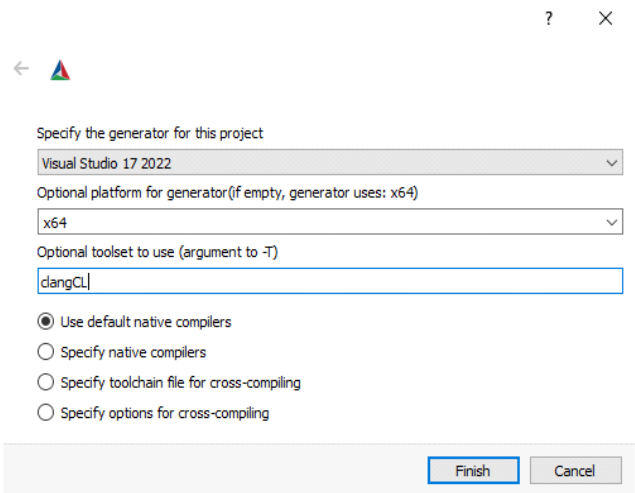


Figure 11. Specify Generator

6. Update the options based on the project requirements. All the available options are listed in [Table 30](#). Select the available and recommended options as follows:

Search:	
Name	Value
AMDCPU_TARGETS	
AOCLSPARSE_CONFIG_DIR	\$(CPACK_PACKAGING_INSTALL_PREFIX)/lib
AOCL_BLI5_INCLUDE_DIR	C:/Program Files/AMD/AOCL_Windows/amd-bli5/include
AOCL_BLI5_LIB	C:/Program Files/AMD/AOCL_Windows/amd-bli5/lib/AOCL-LibBli5-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_ILP64	<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 checked="" type="checkbox"/>
VALGRIND	<input type="checkbox"/>

Figure 12. AOCL-Sparse CMake Config Options

Note: Currently, only single-threaded builds are supported on Windows; multi-threaded support will be added in future release.

7. Click the **Generate** button and then **Open Project**.
8. Complete the following steps in Microsoft Visual Studio GUI:
 - a. Open the AOCL-Sparse Visual Studio project from the build folder using the *aoclsparse.sln* file or the **Open Project** button in CMake GUI.
 - b. 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`.

11.1.2.2 Using Windows Command-line

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

1. Install AOCL-BLAS, AOCL-LAPACK, and AOCL-Utils.

2. Define the environment variable AOCL_ROOT to point to the AOCL Libs installation:

```
set "AOCL_ROOT=C:\Program Files\AMD\AOCL-Windows"
```

For the cases where AOCL_ROOT cannot be exported by placing AOCL-BLAS, AOCL-LAPACK, and AOCL-Utils libraries in the same path, define the following variables during CMake configuration to point to the corresponding libraries and headers:

- AOCL_BLIS_LIB
- AOCL_LIBFLAME
- AOCL_UTILS_LIB
- AOCL_BLIS_INCLUDE_DIR
- AOCL_LIBFLAME_INCLUDE_DIR
- AOCL_UTILS_INCLUDE_DIR

3. Checkout the AOCL-Sparse directory:

```
cd aocl-sparse
```

4. Configure the project along with the following options depending on the required build. For CMake configure options, refer to Table 23. When AOCL_ROOT is defined:

```
cmake -S . -B out_sparse -T clangcl -G "Visual Studio 17 2022" -DCMAKE_CXX_COMPILER=clang-cl -DCMAKE_INSTALL_PREFIX="<aoclsparse_install_path>"
```

When CMake variables are used to define AOCL-BLAS, AOCL-LAPACK, and AOCL-Utils libraries/header:

```
cmake -S . -B out_sparse -T clangcl -G "Visual Studio 17 2022" -DCMAKE_CXX_COMPILER= clang-cl -DCMAKE_INSTALL_PREFIX="<aoclsparse_install_path>" -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" -DAOCL_UTILS_LIB="Utils/Library/with/path" -DAOCL_UTILS_INCLUDE_DIR="path/to/Utils/Headers"
```

Refer to [Table 30](#) to update the parameter options in the command according to the project requirements.

Note: *Currently, only single-threaded builds are supported for Windows; multi-threaded support will be added in future release.*

5. Export the paths for AOCL-BLAS, AOCL-LAPACK, and AOCL-Utils libraries:

```
set "PATH=C:\Users\Program Files\AMD\AOCL-Windows\amd-blis\lib\LP64;%PATH%"
set "PATH=C:\Users\Program Files\AMD\AOCL-Windows\amd-libflame\lib\LP64;%PATH%"
set "PATH=C:\Users\Program Files\AMD\AOCL-Windows\amd-utils\lib;%PATH%"
```

6. Build the project:

```
cmake --build out_sparse --config Release --target install --verbose
```

7. Install AOCL-Sparse:

```
cmake --build out_sparse/ --target install
```


11.1.3 Using Pre-built Libraries

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>

Note:

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

11.2 Usage

The library includes sample programs demonstrating the usage of AOCL-Sparse APIs and they can be used as a starting point to build your application, executable binaries to perform benchmarking and unit test to check the correctness of the library build. These are located in *tests/examples*, *tests/benchmarks* and *tests/unit_tests* directories respectively.

11.2.1 Use by Applications on Linux

To use AOCL-Sparse in your application, compile your source with C or C++ compiler including the AOCL-Sparse header files (`-I$SPARSE_ROOT/include`). If 64-bit integers are used, define the appropriate macro (`-Daoclsparse_ILP64`). Afterwards, link the objects with C++ linker to the AOCL-Sparse library, dependencies (`libflame`, `libblis`, and `libaoclutils` in that order), and `pthread` library. You must match the size of integers for the dependency libraries (LP64 versus ILP64). If multi-threaded library is used, appropriate compiler flags must be defined and the corresponding dependent libraries must be linked. Either static or shared libraries can be used. In the case of shared libraries, you must set their location in `LD_LIBRARY_PATH` before running the resulting executables.

The following sections provide commands to manually build two sample programs from AOCL-Sparse examples directory (*tests/examples*) in two different scenarios.

11.2.1.1 Link to Single-Threaded, LP64, Static Sparse Library

```
export AOCL_ROOT=/opt/aocl
export SPARSE_ROOT=<aoclsparse_install_path>
export LD_LIBRARY_PATH=$AOCL_ROOT/lib_LP64:$LD_LIBRARY_PATH

g++ sample_spmv.cpp -I$SPARSE_ROOT/include -I$AOCL_ROOT/include_LP64 -I$AOCL_ROOT/include_LP64/
alci/.. $SPARSE_ROOT/lib/libaoclsparse.a -Wl,-rpath,$AOCL_ROOT/lib_LP64 $AOCL_ROOT/
lib_LP64/libflame.so $AOCL_ROOT/lib_LP64/libblis.so $AOCL_ROOT/lib_LP64/libaoclutils.so
-lpthread -o sample_spmv

g++ sample_spmv_c.c -I$SPARSE_ROOT/include -I$AOCL_ROOT/include_LP64 -I$AOCL_ROOT/include_LP64/
alci/.. $SPARSE_ROOT/lib/libaoclsparse.a -Wl,-rpath,$AOCL_ROOT/lib_LP64 $AOCL_ROOT/
lib_LP64/libflame.so $AOCL_ROOT/lib_LP64/libblis.so $AOCL_ROOT/lib_LP64/libaoclutils.so
-lpthread -o sample_spmv_c

./sample_spmv
./sample_spmv_c
```

11.2.1.2 Link to Multi-Threaded, ILP64, Shared Sparse Library

```
export AOCL_ROOT=/opt/aocl
export SPARSE_ROOT=<aoclsparse_install_path>
export LD_LIBRARY_PATH=$AOCL_ROOT/lib_ILP64:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH=$SPARSE_ROOT/lib:$LD_LIBRARY_PATH

g++ sample_spmv.cpp -I$SPARSE_ROOT/include -I$AOCL_ROOT/include_ILP64 -I$AOCL_ROOT/
include_ILP64/alci/.. -Daoclsparse_ILP64 -fopenmp $SPARSE_ROOT/lib/libaoclsparse.so -
Wl,-rpath,$AOCL_ROOT/lib_ILP64 $AOCL_ROOT/lib_ILP64/libflame.so $AOCL_ROOT/lib_ILP64/
libblis.so $AOCL_ROOT/lib_ILP64/libaoclutils.so -lpthread -o sample_spmv

g++ sample_spmv_c.c -I$SPARSE_ROOT/include -I$AOCL_ROOT/include_ILP64 -I$AOCL_ROOT/
include_ILP64/alci/.. -Daoclsparse_ILP64 -fopenmp $SPARSE_ROOT/lib/libaoclsparse.so -
Wl,-rpath,$AOCL_ROOT/lib_ILP64 $AOCL_ROOT/lib_ILP64/libflame.so $AOCL_ROOT/lib_ILP64/
libblis.so $AOCL_ROOT/lib_ILP64/libaoclutils.so -lpthread -o sample_spmv_c

./sample_spmv
./sample_spmv_c
```

To build all the example programs using CMake build system, refer to *README.md* in *tests/examples*.

11.2.2 Use by Application on Windows

Complete the following steps to use by application on Windows:

1. Move to installed examples directory:

```
cd <install_dir>\examples
```

2. Define the environment variable AOCL_ROOT to point to AOCL libs installation that has AOCL BLAS, AOCL LAPACK and AOCL UTILS libraries:

```
set "AOCL_ROOT=C:\Program Files\AMD\AOCL-Windows"
```


3. Define SPARSE_ROOT to the AOCL-Sparse package installation path:

```
set "SPARSE_ROOT=%HOME%\amd\aocl-sparse"
```

4. Add AOCL-Sparse, AOCL-BLAS, AOCL-LAPACK and AOCL-Utils library paths to the environment path variable:

```
set "PATH=%SPARSE_ROOT%\lib;%PATH%"
set "PATH=%AOCL_ROOT%\amd-blis\lib\ILP64;%PATH%"
set "PATH=%AOCL_ROOT%\amd-libflame\lib\ILP64;%PATH%"
set "PATH=%AOCL_ROOT%\amd-utils\lib;%PATH%"
```

5. Configure the build system to compile sample applications. Ensure that the build configuration is same as the one used to build the AOCL-Sparse library for the variable BUILD_ILP64:

```
cmake -S . -B out_sparse -G "Visual Studio 17 2022" -T "clangcl" -DCMAKE_CXX_COMPILER=clang-cl
-DBUILD_SHARED_LIBS=ON -DBUILD_ILP64=OFF -DSUPPORT_OMP=OFF
```

6. Compile the sample applications:

```
cmake --build out_sparse
```

7. Run the application:

```
.\out_sparse\sample_spmv.exe
```

Notes:

1. *Currently, only single-threaded builds are supported for Windows; multi-threaded support will be added in future release. So, by default, SUPPORT_OMP=OFF.*
2. *The environment variable "SPARSE_ROOT" takes precedence when searching for sparse library and headers. If the environment variable "SPARSE_ROOT" is not defined, SPARSE_ROOT inherits the path from AOCL_ROOT and looks for AOCL-Sparse installation in AOCL_ROOT. If the sparse library/headers are not found neither in SPARSE_ROOT nor in AOCL_ROOT, an error is returned.*

11.2.3 Performance Benchmarking on Linux

The AOCL-Sparse benchmark executable (called aoclsparse-bench) accepts various input parameters including matrix data and triggers the desired operation while measuring the API performance. Optionally, it can also check the results against its reference implementation. The matrix data can be randomly generated or read from the Matrix Market format (.mtx) input file. The MTX inputs can be downloaded from the SuiteSparse Matrix Collection website (<https://sparse.tamu.edu/>). Usage of both the input types is as follows:

1. Enable BUILD_CLIENTS_BENCHMARKS during AOCL-Sparse installation process (refer to the section 11.1.1).

2. Export paths to the dependent libraries in LD_LIBRARY_PATH if building shared library of AOCL-Sparse:

```
export AOCL_ROOT=/opt/aocl
export SPARSE_ROOT=<aoclsparse_install_path>
export LD_LIBRARY_PATH=$AOCL_ROOT/lib_ILP64:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH=$SPARSE_ROOT/lib:$LD_LIBRARY_PATH
```

3. Navigate to the folder containing the AOCL-Sparse executable:

```
cd out_sparse/tests/staging/
```

4. Run the benchmark:

Random Data:

```
./aoclsparse-bench --function=optmv --precision=d --size=1000 --size=1000 --sizennz=4000 -
verify=1
```

MTX Input:

```
./aoclsparse-bench --function=optmv --precision=d --mtx=LFAT5.mtx --verify=1
```

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

```
export AOCLSPARSE_NUM_THREADS=4
./aoclsparse-bench --function=csrsv --precision=d --size=1000 --size=1000 --sizennz=4000 --
verify=1
```

Note: Running *aoclsparse-bench* without any arguments will provide overview of its parameters.

11.2.4 Performance Benchmarking on Windows

Complete the following steps to run performance benchmarking on Windows:

1. Enable BUILD_CLIENTS_BENCHMARKS during AOCL-Sparse installation process (refer to the section 11.1.1).
2. Define the environment variable AOCL_ROOT to point to AOCL libs installation that has AOCL BLAS, AOCL LAPACK and AOCL UTILS libraries:

```
set "AOCL_ROOT=C:\Program Files\AMD\AOCL-Windows"
```

3. Define SPARSE_ROOT to the AOCL-Sparse package installation path:

```
set "SPARSE_ROOT=%HOME%\amd\aocl-sparse"
```

4. Add AOCL-Sparse, AOCL-BLAS, AOCL-LAPACK and AOCL-Utils library paths to the environment path variable:

```
set "PATH=%SPARSE_ROOT%\lib;%PATH%"
set "PATH=%AOCL_ROOT%\amd-blis\lib\ILP64;%PATH%"
set "PATH=%AOCL_ROOT%\amd-libflame\lib\ILP64;%PATH%"
set "PATH=%AOCL_ROOT%\amd-utils\lib;%PATH%"
```

5. Navigate to the folder containing the AOCL-Sparse executable:

```
cd out_sparse\tests\staging\
```


6. Run the benchmark:

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
```

Note: *Currently, only single-threaded builds are supported for Windows; multi-threaded support will be added in future release.*

11.2.5 Running the Test Suite

The AOCL-Sparse library comes with a comprehensive set of tests which you might want to run when you build the library from source (refer to the sections 11.1.1 and 11.1.2). During the configuration, enable `BUILD_UNIT_TESTS`, `BUILD_CLIENTS_BENCHMARKS` and `BUILD_CLIENTS_SAMPLES`. CMake will then generate all unit tests, examples, and a selection of benchmarking tests as a part of ctests. They can be executed from the command prompt as follows:

```
cd <build_directory>  
ctest -VV
```

Refer to the ctests help on all the possible ways to trigger an individual test or a specific selection.

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. It has multiple implementations of each function, supporting AVX2, AVX512, and ERMS CPU features. The default choice is the best-fit implementation based on the underlying micro-architectural support for CPU features and instructions. It also supports tunable build under which a specific implementation can be chosen for `mem*` functions as per the application requirements with respect to alignments, instruction choice, and threshold values as tunable parameters.

This release of the AOCL-LibMem library supports the following functions:

- `memcpy`
- `mempcpy`
- `memmove`
- `memset`
- `memcmp`
- `strcpy`
- `strncpy`
- `strcmp`
- `strncmp`
- `strlen`

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. Configure for one of the following builds as required:

– GCC

```
# Default Native Build
$ cmake -D CMAKE_C_COMPILER=gcc -S <source_dir> -B <build_dir>

# Cross Compiling AVX2 Binary on AVX512 Machine
$ cmake -D CMAKE_C_COMPILER=gcc -D ALMEM_ARCH=avx2 -S <source_dir> -B <build_dir>
# Cross Compiling AVX512 Binary on AVX2 Machine
$ cmake -D CMAKE_C_COMPILER=gcc -D ALMEM_ARCH=avx512 -S <source_dir> -B <build_dir>

# Enabling Tunable Parameters
$ cmake -D CMAKE_C_COMPILER=gcc -D ENABLE_TUNABLES=Y -S <source_dir> -B <build_dir>
```

– AOCC (Clang)

```
# Default Native Build
$ cmake -D CMAKE_C_COMPILER=clang -S <source_dir> -B <build_dir>

# Cross Compiling AVX2 Binary on AVX512 Machine
$ cmake -D CMAKE_C_COMPILER=clang -D ALMEM_ARCH=avx2 -S <source_dir> -B <build_dir>
# Cross Compiling AVX512 Binary on AVX2 Machine
$ cmake -D CMAKE_C_COMPILER=clang -D ALMEM_ARCH=avx512 -S <source_dir> -B <build_dir>

# Enabling Tunable Parameters
$ cmake -D CMAKE_C_COMPILER=clang -D ENABLE_TUNABLES=Y -S <source_dir> -B <build_dir>
```

4. Build:

```
$ cmake --build <build_dir>
```

5. Install:

```
$ cmake --install <build_dir>

## For custom install path, run configure with "CMAKE_INSTALL_PREFIX"
```

Note: Both shared (*libaocl-libmem.so*) and static (*libaocl-libmem.a*) library files are installed under *<build_dir>/lib/* path. *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/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/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 24.

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 24. 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

Table 24. Application Implementations

Application Requirement	LIBMEM_OPERATION	Instructions	Side-effects
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
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/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 25. 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/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 type:

- Advanced Encryption Standard (AES) block and ChaCha20 stream ciphers
- Secure Hash Algorithms (SHA-2 and SHA-3)
- Cipher, Hash, and Poly1305 based Message Authentication Code (MAC)
- Elliptic-curve Diffie–Hellman (ECDH) and Rivest, Shamir, and Adleman (RSA) key generation functions

The AOCL-Cryptography library has the following functions:

- AES block cipher encrypt/decrypt routines for the following 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)
 - Synthetic Initialization Vector (SIV)
- Stream cipher encrypt/decrypt routine — Chacha20
- SHA-2 digest routines for the following schemes:
 - SHA2_224
 - SHA2_256
 - SHA2_384
 - SHA2_512
 - SHA2_512_224
 - SHA2_512_256
- SHA-3 digest routines for the following schemes:
 - SHA3_224, SHA3_256, SHA3_384, and SHA3_512
 - SHAKE128 and SHAKE256

- MAC routines:
 - Hash-based Message Authentication Code (HMAC)
 - Cipher-based Message Authentication Code (CMAC)
 - Poly1305
- ECDH x25519 key exchange functions:
 - Generate Public Key
 - Compute Secret Key
- RSA 1024/2048
 - Encrypt with public Key
 - Decrypt with private Key

Notes:

1. *Only OAEP-padded mode is supported in AOCL 4.2 release.*
2. *RSA functions are supported in all AMD “Zen” architectures. Architectures other than AMD “Zen” might have partial support.*

13.1 Requirements

- CMake 3.21
- GCC 11.1.0 through 13.1.0
- OpenSSL v3.0.0 through 3.0.5
- Clang 15 on Windows
- [AOCL-Utills library](#)
- AOCC 4.1 or later
- 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 26. 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_UTILS (ON/OFF)	Enable CPUID feature using the AOCL-Utils library
AOCL_UTILS_INSTALL_DIR	AOCL-Utils installation path
OPENSSL_INSTALL_DIR	OpenSSL (3.0.0 through 3.0.5) installation path
CMAKE_INSTALL_PREFIX	AOCL-Cryptography installation path
ALCP_SANITIZE (ON/OFF)	Enable sanitizers (asan, ubsan, msan, 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"> 1. The IPP header files should be added to the <code>CPLUS_INCLUDE_PATH</code> environment variable (working version for IPP is 2021_8). 2. OpenSSL provider support is not yet enabled for ECDH, RSA, and SIV functions. 3. IPP provider support is not yet enabled for ECDH and RSA functions.
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`

9. To execute tests/benchmarks using KAT framework, run the following commands:

```
git-lfs fetch
git-lfs checkout
```

For detailed steps to execute KAT tests/bench, refer to [tests_Readme](#) and [bench_Readme](#) files respectively.

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 through 3.0.5) 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 27. AOCL-Cryptography - Windows Options

Option	Description
-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
```

7. To execute tests/benchmarks using the KAT framework, run the following commands:

```
git-lfs fetch
git-lfs checkout
```

For detailed steps to execute KAT tests/bench, refer to [tests_Readme](#) and [bench_Readme](#) files respectively.

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

For using the cipher routines, use the header file:

include/alcp/cipher.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. `export LIBRARY_PATH=<path to aocl crypto lib>:<path to aocl utils lib>:<path to openssl lib>:$LIBRARY_PATH;`
4. `make`
5. To run example applications (for digest):

```
LD_LIBRARY_PATH=<path to aocl crypto lib>:<path to aocl utils 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.
- OpenMP based multi-threaded implementation of lz4, zlib, zstd, and snappy compression 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.
- Instruction set dispatch, running non-optimized code, and log level selection using environment variables at runtime:
 - Instruction Set Dispatch (SSE2, AVX, AVX2, and AVX512)
 - Enabling of logging and selection of log level
- Non-optimized (reference) code executionA 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. Also, it supports the testing of the compression methods through their native APIs directly, offers memory checks using Valgrind, ASAN, and source code coverage using GCOV.
- 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.
3. To build the library with multi-threaded support, set the CMake build option `AOCL_ENABLE_THREADS=ON`. The library uses OpenMP for multi-threaded support. Maximum numbers of threads to use can be set using the environment variable `OMP_NUM_THREADS`. To run the library built with `AOCL_ENABLE_THREADS=ON` in single threaded mode, set `OMP_NUM_THREADS=1`.

Refer to the section “*Optional Optimization Options*” on page 148 for the complete list of supported CMake build options.

- It is recommended that applications use the unified APIs of the library over the native APIs for ease of integration and minimal code modifications required for addition of new compression methods.

14.2 Running AOCL-Compression Test Bench on Linux

Test bench supports several options to validate, benchmark, or debug the supported compression methods.

It can be configured to use the unified APIs or native APIs to invoke 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*, build the library by using ``-DAOCL_ENABLE_LOG_FEATURE=ON`` and set the environment variable ``AOCL_ENABLE_LOG`` to any of the following:

- ``AOCL_ENABLE_LOG=ERR`` for Error logs
- ``AOCL_ENABLE_LOG=INFO`` for Error, Info logs
- ``AOCL_ENABLE_LOG=DEBUG`` for Error, Info, and Debug logs
- ``AOCL_ENABLE_LOG=TRACE`` for Error, Info, Debug, and Trace logs

When building the library for best performance, do not enable

``DAOCL_ENABLE_LOG_FEATURE``. To run the test bench using native APIs, use the `-n` option. An example to run the test bench and validate the outputs (from all the supported compression and decompression methods) for a given input file using the native APIs:

```
aocl_compression_bench -a -n -t <input filename>
```

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 [the README file](https://github.com/amd/aocl-compression) available with the source package in GitHub (<https://github.com/amd/aocl-compression>).

Note: *Library portability on Windows is limited to the systems with support for AVX2 instruction set or later.*

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          */
} aocl_compression_desc;

```


14.4.3 Library Return Error Codes

```
typedef enum
{
    ERR_INVALID_INPUT = -5,           ///

```

14.4.4 Multi-threaded API Set

```
//Interface API to get the upper bound of RAP frame bytes that will be added during
    multithreaded compression.
int32_t aocl_get_rap_frame_bound_mt(void);

//Interface API to get the length of the RAP frame in the compressed stream
int32_t aocl_skip_rap_frame_mt(char* src, int32_t src_size);
```

14.4.5 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
);

```

Note: When calling LZ4HC native APIs that use external dictionary such as `LZ4_compress_HC_extStateHC()`, it is recommended to set the environment variable `AOCL_DISABLE_OPT=ON`, due to a known issue.

14.4.6 Example Program

14.4.6.1 Single-threaded APIs

The following example 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->optOff = 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
```

14.4.6.2 Multi-threaded APIs

When a library is built with multi-threaded support (refer to section 14.1.2), a Random Access Point (RAP) frame is added at the start of the compressed stream to support parallel decompression of the compressed stream/file. You must allocate sufficient additional bytes in the destination buffer to account for this frame.

A stream compressed with multi-threaded AOCL-Compression library can be decompressed using any single-threaded standard decompressor by skipping the initial block of bytes containing the RAP frame present at the start of the stream.

Following test program shows the sample usage and calling sequence of AOCL-Compression APIs to get an ST compatible compressed stream from the stream produced by AOCL MT compressor:

```
<code-section-here-start>
#include <stdio.h>
#include <stdlib.h>
#include "aocl_compression.h"
#include "aocl_threads.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);

    aocl_compression_type method = LZ4; // One of the compression methods as per
aocl_compression_type
    aocl_compression_handle->level = 0;
    aocl_compression_handle->optVar = 0;
    aocl_compression_handle->optOff = 0;
    aocl_compression_handle->measureStats = 0;
    aocl_compression_handle->inSize = file_size;
    aocl_compression_handle->outSize = (file_size + (file_size / 6) + (16 * 1024)) /* LZ4 ST
compress bound */
                                     + aocl_get_rap_frame_bound_mt() /* upper bound of RAP
frame bytes */;
    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. MT compress
resultComp = aocl_llc_compress(aocl_compression_handle, method);

if (resultComp <= 0)
{
    printf("Compression: failed\n");
    goto error_exit;
}
printf("Compression: done\n");
//3. ST decompress
// Get number of bytes for the RAP frame
int rap_frame_len = aocl_skip_rap_frame_mt((char *)compPtr, resultComp);

// Skip RAP frame in input stream and pass this to ST decompressor
aocl_compression_handle->inSize = resultComp - rap_frame_len;
aocl_compression_handle->outSize = file_size;
aocl_compression_handle->inBuf = compPtr + rap_frame_len;
aocl_compression_handle->outBuf = decompPtr;

// Pass format compliant stream to aocl decompressor (or any legacy ST decompressor)
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 program on a Linux system using GCC or AOCC, you must specify the path to *aocl_compression.h* and *aocl_threads.h* header files and link with *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 28. Optional Optimization Options

Option	Description
AOCL_ENABLE_THREADS	Enable multi-threaded compression and decompression using SMP based OpenMP threads (Disabled by default)
AOCL_ENABLE_LOG_FEATURE	Enables logging through environment variable `AOCL_ENABLE_LOG` (Disabled by default)
BUILD_DOC	Build documentation for this library (Disabled by default)
AOCL_TEST_COVERAGE	Enable GTest and AOCL test bench based CTest suite (Disabled by default)
CODE_COVERAGE	Enable source code coverage, supported only on Linux with the GCC compiler (Disabled by default)
ASAN	Enable Address Sanitizer checks. Only supported on Linux/Debug build (Disabled by default)
VALGRIND	Enable Valgrind checks. Only supported on Linux/Debug and incompatible with ASAN=ON (Disabled by default)
LZ4_FRAME_FORMAT_SUPPORT	Enable building LZ4 with Frame format and API support (Enabled by default)
AOCL_LZ4_OPT_PREFETCH_BACKWARDS	Enable LZ4 optimizations related to backward prefetching of data (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. Preferred to be used with Silesia corpus (Disabled by default)
AOCL_LZ4_NEW_PRIME_NUMBER	Enable the usage of a new prime number for LZ4 hashing function. Preferred to be used with Silesia corpus (Disabled by default)
AOCL_LZ4_EXTRA_HASH_TABLE_UPDATES	Enable storing of additional potential matches to improve compression ratio. Recommended for higher compressibility use cases (Disabled by default)

Table 28. Optional Optimization Options

Option	Description
AOCL_LZ4_HASH_BITS_USED	Control the number of bits used for LZ4 hashing, allowed values are LOW (low perf gain and less CR regression) and HIGH (high perf gain and high CR regression) (Disabled by default)
AOCL_LZ4HC_DISABLE_PATTERN_ANALYSIS	Disable Pattern Analysis in LZ4HC for level 9 (Enabled by default)
SNAPPY_MATCH_SKIP_OPT	Enable Snappy match skipping optimization (Disabled by default)
ZLIB_DEFLATE_FAST_MODE	Enable ZLIB deflate quick strategy (Disabled by default)
AOCL_ZSTD_SEARCH_SKIP_OPT_FAST	Enable ZSTD match skipping optimization, and reduce search strength/tolerance for levels 1-4 (Disabled by default)
AOCL_ZSTD_WILDCOPY_LONG	Faster wildcopy when match lengths are long in ZSTD decompression (Disabled by default)
AOCL_XZ_UTILS_LZMA_API_EXPERIMENTAL	Build with xz utils lzma APIs. Experimental feature with limited API support (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.

AOCL-Utills 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
 - Cache size and line size
 - Number of ways and sets
 - Number of logical processors sharing cache
 - Number of physical partitions
 - Fully associative
 - Self-initializing
 - Cache Inclusive/Exclusive

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
- AOCC 4.1 or later
- Refer to *“Build Utilities” on page 18* for Make and Microsoft Visual Studio versions
- For more information on the supported operating systems, refer to *“Operating Systems” on page 17*

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 19*.

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-Utils 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-Utils.

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>
```

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.

15.3.6 Integrate with Other Libraries/Applications

Following are the build systems to integrate in library/application with AOCL-Utils:

- CMAKE:

In the CMake file, use the following:

- TARGET_INCLUDE_DIRECTORIES() – path of libaoclutils include directory
- TARGET_LINK_LIBRARIES() – path to link libaoclutils binaries

- Make:

In the compiler flags of Make file, use the following:

- “-I” - path of libaoclutils include directory
- “-l, -L” - path to link libaoclutils binaries

Chapter 16 AOCL Tuning Guidelines

This section provides tuning recommendations for AOCL.

16.1 AOCL-BLAS Thread Control

Application can set the desired number of threads during AOCL-BLAS initialization and runtime as explained below.

16.1.1 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 29. 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.

16.1.2 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 30. Sample Scenarios - 2

Sample Pseudo Code for Application	Sample Command Executed	m	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 < 500) omp_set_num_threads(8); if(m >= 500) omp_set_num_threads(16); if(m >= 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)

16.1.2.1 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: From AOCL 4.1, support for calling AOCL-BLAS within nested OpenMP parallelism has been improved. Hence, using the standard OpenMP mechanisms should be sufficient for most of the use cases.

16.2 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 31. 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	Unset	Number of threads determined by AOCL Dynamic.
Enabled	Set	Minimum of BLIS_NUM_THREADS or the number of threads determined by AOCL.

16.2.1 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, DNRM2, DZNRM2, 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
- Based on the input parameters (such as size, transpose, and storage format), optimal code path for the given number of threads would be executed. This can be single-threaded even if the number of threads set is more than 1.

16.3 AOCL-BLAS DGEMM Multi-thread Tuning

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.

16.3.1 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.

16.3.2 Architecture Specific Tuning

16.3.2.1 2nd and 3rd Gen AMD EPYC™ Processors

To achieve the best DGEMM multi-thread performance on 2nd Gen AMD EPYC™ processors (codenamed "Rome") and 3rd 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
```


16.3.2.2 1st Gen AMD EPYC™ Processors

To achieve the best DGEMM multi-thread performance on the 1st 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 26.
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 26.
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
```

16.4 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_blkksz_init_easy( &blkzs[ BLIS_MC ], 144, 240, 144, 72 );
bli_blkksz_init_easy( &blkzs[ BLIS_KC ], 256, 512, 256, 256 );
bli_blkksz_init_easy( &blkzs[ 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_blkksz_init_easy( &blkzs[ BLIS_MC ], 144, 510, 144, 72 );
bli_blkksz_init_easy( &blkzs[ BLIS_KC ], 256, 1024, 256, 256 );
bli_blkksz_init_easy( &blkzs[ BLIS_NC ], 4080, 4080, 4080, 4080 );
```

16.5 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.
```

16.6 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=ON` or `ENABLE_AMD_AOCC_FLAGS=ON`. 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

16.7 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-openssl` 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 169.
- 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 17 Support

For support options, the latest documentation, and downloads refer to AMD Developer Central (<https://www.amd.com/en/developer/aocl.html>).

Chapter 18 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 1st Gen AMD EPYC™ Processors (codenamed “Naples”), CPU Core AMD “Zen”
 - CPU Family: 23
 - Model: Values in the range <1 – 47>
- b. For 2nd Gen AMD EPYC™ Processors (codenamed “Rome”), CPU Core AMD “Zen2”
 - CPU Family: 23
 - Model: Values in the range <48 – 63>
- c. For 3rd Gen AMD EPYC™ Processors (codenamed “Milan”), CPU Core AMD “Zen3”
 - CPU Family: 25
 - Model: Values in the range <1 – 15>
- d. For 4th 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 1st Gen AMD EPYC™ Processors (codenamed “Naples”), CPU Core AMD “Zen”
 - CPU Family: 23
 - Model: Values in the range <1 – 47>
- b. For 2nd Gen AMD EPYC™ Processors (codenamed “Rome”), CPU Core AMD “Zen2”
 - CPU Family: 23
 - Model: Values in the range <48 – 63>

- c. For 3rd Gen AMD EPYC™ Processors (codenamed “Milan”), CPU Core AMD “Zen3”
 - CPU Family: 25
 - Model: Values in the range <1 – 15>
- d. For 4th 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.