AOCL User Guide

Publication # **57404**	Revision # **4.0**
Issue Date **November 2022**	

Contents

List of Tables

List of Figures

Revision History

Date	Revision	Description
November 2022	4.0	• 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	• 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^{TM}, Ryzen^{TM} Threadripper^{TM}, and Ryzen^{TM}. This document provides instructions on installing and using all the AMD optimized libraries.

AOCL is comprised of the following libraries:

- **AOCL-BLIS (BLAS Library)** is a portable software framework for performing high-performance Basic Linear Algebra Subprograms (BLAS) functionality.
- **AOCL-libFLAME (LAPACK)** is a portable library for dense matrix computations that provides the functionality present in the Linear Algebra Package (LAPACK).
- **AOCL-FFTW (Fastest Fourier Transform in the West)** is a comprehensive collection of fast C routines for computing the Discrete Fourier Transform (DFT) and various special cases.
- **AOCL-LibM (AMD Core Math Library)** is a software library containing a collection of basic math functions optimized for x86-64 processor based machines.
- **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 processors, including EPYC^{TM}, Ryzen^{TM} Threadripper^{TM} PRO, and Ryzen^{TM}.
- **AOCL-LibMem** is AMD’s optimized implementation of memory/string functions.
- **AOCL-Cryptography** is AMD’s optimized implementation of cryptographic functions (AES Encryption/Decryption and SHA2 Digest).
- **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, AOCL-Cryptography, and AOCL-Compression.

Additionally, AMD provides Spack (*https://spack.io/*) recipes for installing AOCL-BLIS, AOCL-libFLAME, AOCL-ScaLAPACK, AOCL-LibM, AOCL-FFTW, and AOCL-Sparse libraries.

For more information on the AOCL release and installers, refer the AMD Developer Central (*https://developer.amd.com/amd-aocl/*).

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 Supported OS and Compilers

This section lists the supported operating systems, compilers, and prerequisites for AOCL 4.0. It 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 21.04
- CentOS 7 and 8
- Red Hat® Enterprise Linux® (RHEL) 8.2, 8.3.1, and 8.6
- SUSE Linux Enterprise Server (SLES) 15 SP3
- Windows Server 2019 and 2022
- Windows® 10
- Windows 11 Pro

2.2 Compilers

- GCC 9.1.0, 9.2.1, and 11.2
- AOCC 3.1, 3.2, and 4.0
- LLVM™ 13 and 14

2.3 Library

Glibc 2.17 and 2.31

2.4 Message Passing Interface (MPI)

Open MPI 4.1.4

2.5 Programming Language

- Python versions 2.7,2.8, 3.4, and 3.6
- Perl 5.14 and 5.34

2.6 Build Utilities

- GNU Make 4.2
- CMake 3.19.6 and 3.23.3
- Microsoft Visual Studio 2019 (build 16.8.7)/2022 (build 17.3.2)

Chapter 3 Installing AOCL

3.1 Building from Source

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

- AOCL-BLIS (*https://github.com/amd/blis*)
- AOCL-libFLAME (*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*)

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

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://developer.amd.com/amd-aocl/#download*) section to the target machine.
2. Use the command `tar -xvf <aocl-linux-<compiler>-4.0.tar.gz>` to untar the package.

   The installer file *install.sh* is available in `aocl-linux-<compiler>-4.0`.
3. Run *./install.sh* to install the AOCL package (all libraries) to the default *INSTALL_PATH: /home/<username>/amd/aocl/4.0*.

   Use `install.sh` to print the usage of the script, afew supported options are:

   `-h` — Print the help.

   `-t` — Custom target directory to install libraries.

   `-l` — Library to be installed.

   `-i` — Select LP64/ILP64 libraries to be set as default.

4. 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>`.
5. 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`.

   Note: *For the AOCC package, AOCL-Cryptography is not supported.*
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.

   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-BLIS library in the path */home/amd* and sets LP64 AOCL-BLIS 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://developer.amd.com/amd-aocl/*) to download the individual library binaries from the respective pages.

For example, AOCL-BLIS and AOCL-libFLAME tar packages are available in the BLAS library page (*https://developer.amd.com/amd-aocl/blas-library/).*

3.2.3 Using Debian and RPM Packages

The Debian and RPM packages of AOCL are available in the **Download** section (*https://developer.amd.com/amd-aocl/#download*).

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 4.0 Debian package to the target machine.
2. Check the installation path before installing.

   ```
   $ dpkg -c aocl-linux-gcc-4.0_1_amd64.deb
   ```

3. Install the package.

   ```
   $ sudo dpkg -i aocl-linux-gcc-4.0_1_amd64.deb
   Or
   $ sudo apt install ./aocl-linux-gcc-4.0_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-4.0
   ```

5. List the contents of the package.

   ```
   $dpkg -L aocl-linux-gcc-4.0
   ```

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.

   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>-4.0/*
   - Compiler: aocc or gcc

   For example, INSTALL_PATH for aocc compiler is */opt/AMD/aocl/aocl-linux-aocc-4.0/*.

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

   ```
   cd /opt/AMD/aocl/aocl-linux-<compiler>-4.0/
   sudo bash setenv_aocl.sh <ilp64 / lp64>
   ```

Uninstalling Debian package

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

```
$ sudo dpkg -r aocl-linux-gcc-4.0
or
$ sudo apt remove aocl-linux-gcc-4.0
```

Installing RPM Package

Complete the following steps to install the AOCL RPM package:

1. Download the AOCL 4.0RPM package to the target machine.
2. Install the package.

   ```
   $ sudo rpm -ivh aocl-linux-gcc-4.0-1.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-4.0-1.x86_64
   ```
4. List the contents of the package.

   ```
   $ rpm -ql aocl-linux-gcc-4.0-1
   ```
5. AOCL libraries supportthe 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.

   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>-4.0/*
   - Compiler: aocc or gcc

   For example, INSTALL_PATH for aocc compiler is */opt/AMD/aocl/aocl-linux-aocc-4.0/*.
6. To change the default library path to ILP64 / LP64, use the script as follows:

   ```
   cd /opt/AMD/aocl/aocl-linux-<compiler>-4.0/
   sudo bash setenv_aocl.sh <ilp64 / lp64>
   ```

Uninstalling RPM package

Execute the following command to uninstall the AOCL RPM package:

```
$ rpm -e aocl-linux-gcc-4.0-1
```

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://developer.amd.com/amd-aocl/#download*) 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-BLIS

AOCL-BLIS is a high-performant implementation of the Basic Linear Algebra Subprograms (BLAS). The BLAS was designed to provide the essential ke4rnels of matrix and vector computation and are the most commonly used and computationally intensive operations in dense numerical linear algebra. Select kernels have been optimized for the AMD “Zen”-based processors, for example, AMD EPYC^{TM}, AMD Ryzen^{TM}, AMD Ryzen^{TM} Threadripper^{TM} processors by AMD and others.

AMD offers the optimized version of BLIS (AOCL-BLIS) that supports C, FORTRAN, and C++ template interfaces for the BLAS functionalities.

4.1 Installation on Linux

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

4.1.1 Build AOCL-BLIS from Source

GitHub URL: *https://github.com/amd/blis*

You can use the following ways to build AOCL-BLIS 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-BLIS 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^{TM} (code name "Naples") processor, the ***zen*** configuration will be auto-selected. For a build machine using the 2nd Gen AMD EPYC^{TM} processor (code name "Rome"), the ***zen2*** configuration will be auto-selected. From BLIS 3.0 forward, ***zen3*** will be auto-selected for the 3rd Gen AMD EPYC^{TM} processor (code name "Milan"). From BLIS 4.0 forward, ***zen4*** will be auto-selected for the 4th Gen AMD EPYC^{TM} processor (code name "Genoa").
- **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 32.

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-BLIS. For a complete list of the options and their descriptions, use the command `./configure --help`.

4.1.1.1 Single-thread AOCL-BLIS

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

1. Clone the AOCL-BLIS 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-BLIS

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

1. Clone the AOCL-BLIS git repository(*https://github.com/amd/blis.git*).
2. Configure the library as required:

```
GCC (Default)

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

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

[Mode] values can be openmp and no. "no" will disable multi-threading.
```

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

4.1.1.3 Verifying AOCL-BLIS Installation

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

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

```
$ make check
```

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

```
$ make checkcpp
```

4.1.2 Using Pre-built Binaries

AOCL-BLIS library binaries for Linux are available at the following URLs:

https://github.com/amd/blis/releases

https://developer.amd.com/amd-aocl/blas-library/

Also, the AOCL-BLIS binary can be installed from the AOCL master installer tar file (*https://developer.amd.com/amd-aocl/*).

The master installer includes the following:

- Single threaded and multi-threaded AOCL-BLIS binaries.
- Binaries built with **amdzen** config with LP64 and ILP64 integer support.
- Multi-threaded AOCL-BLIS 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 15.

4.2 Application Development Using AOCL-BLIS

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

4.2.1 API Compatibility Layers (Calling AOCL-BLIS)

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

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

AOCL-BLIS has its own APIs (called BLIS APIs) that provide more flexibility and control to achieve the best performance.

The following table lists all the supported layers:

Table 1. BLIS API Compatibility Layers

API Compatibility Layer	Header Files	Configuration Option	Usages
BLAS (Fortran)	Not applicable	`--enable-blas`	Use this option when calling BLIS from Fortran applications. API Name Format: DGEMM
BLAS (C)	blis.h	`--enable-blas`	Use this option when calling BLIS from C application using BLAS type parameters. API Name Format: dgemm_
CBLAS	cblas.h	`--enable-cblas (Implies --enable-blas)`	Use this option when calling BLIS from C application using CBLAS type parameters. API Name Format: cblas_dgemm
BLIS - C Non Standard	blis.h	Default	This is AOCL-BLIS library specific (non-standard) interface, it provides most flexibility in calling AOCL-BLIS 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-BLIS 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 2. AOCL-BLIS API Compatibility - Advance Options

Feature	Configuration Option	Usages
ILP64 Support	`--blas-int-size=SIZE`	This option can be used to specify the integer types used in external BLAS/CBLAS interfaces. Accepted Values: ILP64 - SIZE = 64 LP64 - SIZE = 32 (Default)
Complex Number return handling	`--complex-return=gnu|intel`	The complex numbers can be returned through registers or the hidden parameter. Based on the way application is calling the API, the library must be configured to match the return value receptions. gnu = return complex values through registers intel = return complex values through hidden parameter. For more information and example, refer “Returning Complex Numbers” on page 31.

4.2.3 Linking Application with AOCL-BLIS

The AOCL-BLIS 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-BLIS-header>  <link-options> -o test_blis.x
```

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

Table 3. AOCL-BLIS Application - Link Options

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

4.2.3.1 Example - Dynamic Linking and Execution

AOCL-BLIS 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-BLIS 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/blis/ -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 Example Application - AOCL-BLIS Usage in FORTRAN

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

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-BLIS 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-BLIS Usage in C

There are multiple ways to use BLIS with an application written in C. While you can always use the native BLIS API, AOCL-BLIS also includes BLAS and CBLAS interfaces.

4.2.5.1 Example Application - Using BLIS with 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 BLIS' 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 BLIS 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/blis/ -lpthread -lm path/to/libblis.a
```

4.2.5.2 Example Application - Using AOCL-BLIS 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-BLIS 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-BLIS, you must provide the flag '--enable-cblas' to the 'configure' command while building the AOCL-BLIS library.*

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

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

4.2.5.3 Returning Complex Numbers

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

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

gfortran Example:

- Configure Option:

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

- API Call:

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

ifort example:

- Configure Option:

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

- API Call:

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

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

4.3 Migrating/Porting

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

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

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-BLIS:

Table 4. Porting to AOCL-BLIS

	MKL	OpenBLAS	AOCL-BLIS	
			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-BLIS Library Features

4.4.1 Dynamic Dispatch

Starting from AOCL 3.1, AOCL-BLIS supports Dynamic Dispatch feature. It enables you to use the same binary on 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 BLIS on a cluster having nodes of different architectures.

Dynamic Dispatch addresses this issue by building a single binary compatiblewith 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”, AND “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-BLIS

Dynamic Dispatch must be enabled while building the AOCL-BLIS library. This is done by building the library for **amdzen** configuration as explained in “Build AOCL-BLIS from Source” on page 20.

Code Path

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-BLIS can be forced to use a specific architecture by setting the environment variable BLIS_ARCH_TYPE as follows:

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

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

You must note the following:

- The code path names are not case sensitive.
- The enumeration number for a given code path may change from release to release as new code paths are added. It has changed in AOCL-BLIS 4.0 from the previous release (3.2).
- 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-BLIS build is being used. This will result in an error message from the AOCL-BLIS library which will then abort.
  - A code path executes instructions unavailable on the processor being used, for example, trying to run the AMD “Zen4” code path (which may use AVX512 instructions) on a AMD “Zen3” or older system. If this happens, the program may stop with an "illegal instruction" error. This may be routine and problem size dependent.

In some circumstances, setting BLIS_ARCH_TYPE incorrectly may cause errors. If you are building AOCL-BLIS 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:

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

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.

4.4.2 BLIS - Running the Test Suite

The AOCL-BLIS source directory contains a test suite to verify the functionality of AOCL-BLIS 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-BLIS 3.1, the dynamic selection of number of threads is supported. If the number of threads are not specified, AOCL-BLIS 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:

```
$ BLIS_NUM_THREADS=2 make test
```

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

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

4.4.3 Testing/Benchmarking

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

The source file for this driver is *test/test_gemm.c* and the executable is *test/test_gemm_blis.x*.

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

Enabling File Inputs

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

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

Building Test Driver

Execute the following commands to build the test driver:

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

Creating an Input File

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

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

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

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

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

- CS_A >= m
- CS_B >= k
- CS_C >= 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 BLIS APIs

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

Table 5. BLIS APIs

API	Usages	
bli_info_get_version_str	Returns the version string in the form of "AOCL-BLIS 4.0.0 Build yyyyddmm".	
bli_info_get_enable_openmp bli_info_get_enable_pthreads bli_info_get_enable_threading	Returns true if OpenMP/ pthreads are enabled and false otherwise.	
bli_thread_get_num_threads[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 Debugging Build Using GDB

The AOCL-BLIS 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-BLIS:

```
$ 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-BLIS-header> <path-to-BLIS-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=0x7fffffffd1c0,
beta=0x2465400 <BLIS_ZERO>,
    c=0x4fe66e <bli_obj_equals+300>, cntx=0x7fffffffb320, cntl=0x0) at kernels/zen/3/
bli_gemm_small.c:110
110     {
(gdb) bt
#0  bli_gemm_small (alpha=0x7fffffffcf40, a=0x2471b30, b=0x7fffffffd1c0, beta=0x2465400
<BLIS_ZERO>,
    c=0x4fe66e <bli_obj_equals+300>, cntx=0x7fffffffb320, cntl=0x0) at kernels/zen/3/
bli_gemm_small.c:110
#1  0x00000000007caab6 in bli_gemm_front (alpha=0x7fffffffd1c0, a=0x7fffffffd120,
b=0x7fffffffd080,
    beta=0x7fffffffcfe0, c=0x7fffffffcf40, cntx=0x2471b30, rntm=0x7fffffffce50, cntl=0x0)
    at frame/3/gemm/bli_gemm_front.c:83
#2  0x00000000005baf42 in bli_gemmnat (alpha=0x7fffffffd1c0, a=0x7fffffffd120,
b=0x7fffffffd080,
    beta=0x7fffffffcfe0, c=0x7fffffffcf40, cntx=0x2471b30, rntm=0x7fffffffce50)
    at frame/ind/oapi/bli_l3_nat_oapi.c:83
#3  0x00000000005474a2 in dgemm_ (transa=0x7fffffffd363 "N\320\a", transb=0x7fffffffd362
"NN\320\a",
    m=0x7fffffffd36c, n=0x7fffffffd364, k=0x7fffffffd368, alpha=0x24733c0, a=0x7ffff53e2040,
lda=0x7fffffffd378,
    b=0x7ffff355d040, ldb=0x7fffffffd374, beta=0x2473340, c=0x7ffff16d8040, ldc=0x7fffffffd370)
    at frame/compat/bla_gemm.c:559
#4  0x0000000000413a1c in main (argc=1, argv=0x7fffffffd988) at test_gemm.c:321
(gdb)
```

4.5.2 Viewing Logs

The AOCL-BLIS library provides Debug and Trace features:

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

The key features of this functionality are as follows:

- Can be enabled/disabled at compile time.

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

4.5.2.1 Function Call Tracing

The function call tracing is implemented using hard instrumentation of the AOCL-BLIS 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.

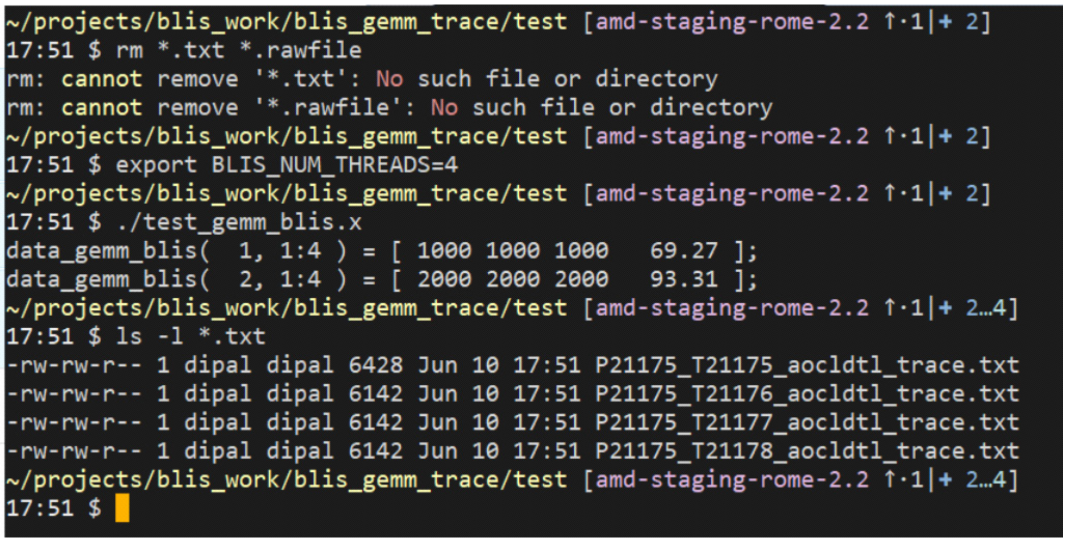
Complete the following steps to enable and view the traces:

1. Enable the trace support as follows:
   a. Modify the source code to enable tracing.
   ```
   Open file <blis 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 <blis 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-BLIS from Source” on page 20.

4. Run the application to generate the trace data.

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

   The following figure shows a sample running the call tracing function using the test_gemm application:



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

Figure 1. Sample Run of Function Call Tracing

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

   P<process id>_T<thread id>_aocldtl_trace.txt

5. View the trace data.

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

4.5.2.2 Debug Logging

The debug logging works very similar to the function call tracing and uses the same infrastructure. However, it can be enabled independent of the trace feature to avoid cluttering of the overall debugging information. This feature is primarily used to print the input values of the BLIS 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 <blis 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 <blis folder>/aocl_dtl/aocldtlcf.h
```

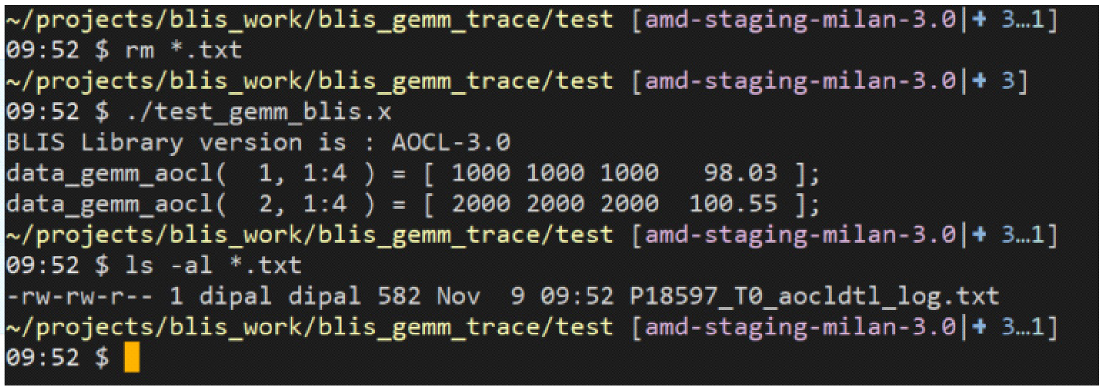
   b. Change the following macro as required. Beginning with Level 5 should be a good compromise in terms of details and resource requirement. The higher the level (maximum is 10), the deeper is the call stack. A lower level reduces the depth of the call stack used for a trace generation.

```
#define AOCL_DTL_TRACE_LEVEL  AOCL_DTL_LEVEL_TRACE_5
```

3. Build the library as explained in “Build AOCL-BLIS from Source” on page 20.
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 BLIS 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 $ 
```

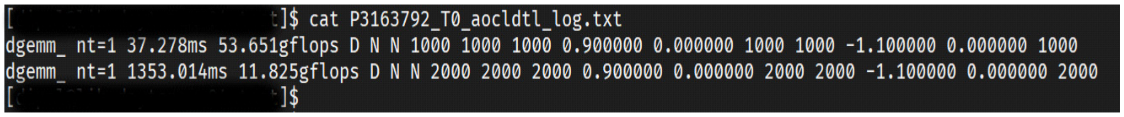
Figure 2. Sample Run with Debug Logs Enabled

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:



```
[[redacted]]$ 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
[[redacted]]$
```

Figure 3. Debug Logs Showing Input Values of GEMM

4.5.2.3 Usages and Limitations

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

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

4.5.3 Checking AOCL-BLIS 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-BLIS and AOCL-libFLAME) periodically updates the application with progress made through a callback function.

Usage

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

Callback Definition

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

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

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

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

Table 6. 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-BLIS

Example

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

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

Register the callback with:

AOCL_BLIS_set_progress(AOCL_progress); // for AOCL-BLIS

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-BLIS, AOCL-libFLAME, and AOCL-ScaLAPACK.

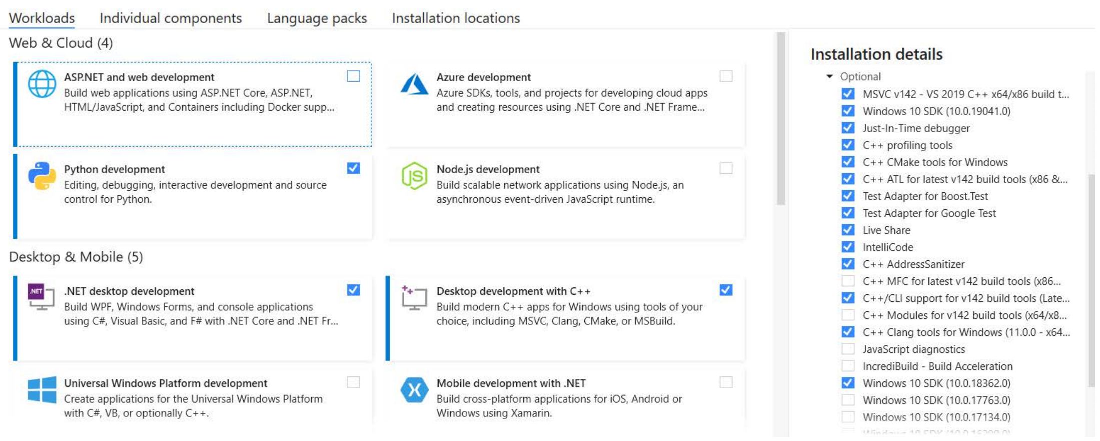
4.6 Build AOCL-BLIS from Source on Windows

GitHub URL: *https://github.com/amd/blis*

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

Prerequisites

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



Web & Cloud (4)

- [ ] **ASP.NET and web development** — Build web applications using ASP.NET Core, ASP.NET, HTML/JavaScript, and Containers including Docker supp...
- [ ] **Azure development** — Azure SDKs, tools, and projects for developing cloud apps and creating resources using .NET Core and .NET Frame...
- [✓] **Python development** — Editing, debugging, interactive development and source control for Python.
- [ ] **Node.js development** — Build scalable network applications using Node.js, an asynchronous event-driven JavaScript runtime.

Desktop & Mobile (5)

- [✓] **.NET desktop development** — Build WPF, Windows Forms, and console applications using C#, Visual Basic, and F# with .NET Core and .NET Fr...
- [✓] **Desktop development with C++** — Build modern C++ apps for Windows using tools of your choice, including MSVC, Clang, CMake, or MSBuild.
- [ ] **Universal Windows Platform development** — Create applications for the Universal Windows Platform with C#, VB, or optionally C++.
- [ ] **Mobile development with .NET** — Build cross-platform applications for iOS, Android or Windows using Xamarin.

Installation details

Optional

- [✓] MSVC v142 - VS 2019 C++ x64/x86 build t...
- [✓] Windows 10 SDK (10.0.19041.0)
- [✓] Just-In-Time debugger
- [✓] C++ profiling tools
- [✓] C++ CMake tools for Windows
- [✓] C++ ATL for latest v142 build tools (x86 &...
- [✓] Test Adapter for Boost.Test
- [✓] Test Adapter for Google Test
- [✓] Live Share
- [✓] IntelliCode
- [✓] C++ AddressSanitizer
- [ ] C++ MFC for latest v142 build tools (x86...
- [✓] C++/CLI support for v142 build tools (Late...
- [ ] C++ Modules for v142 build tools (x64/x8...
- [✓] C++ Clang tools for Windows (11.0.0 - x64...
- [ ] JavaScript diagnostics
- [ ] IncrediBuild - Build Acceleration
- [✓] Windows 10 SDK (10.0.18362.0)
- [ ] Windows 10 SDK (10.0.17763.0)
- [ ] Windows 10 SDK (10.0.17134.0)

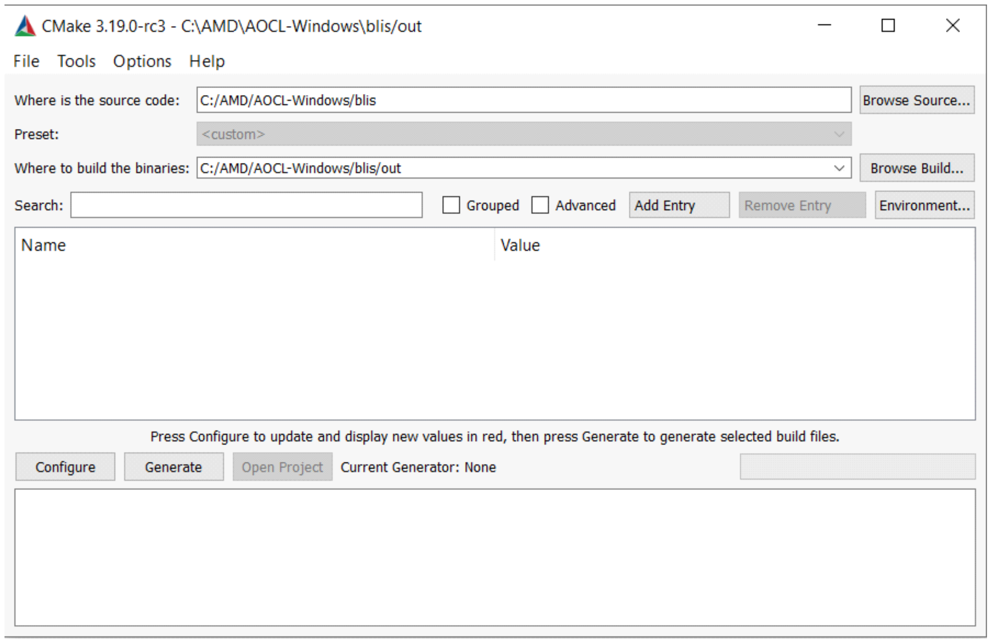
Figure 4. Microsoft Visual Studio Prerequisites

4.6.1 Building AOCL-BLIS 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-BLIS source code) and **build** (folder in which the project files will be generated, for example, **out**) folder paths as shown in the following figure:



CMake 3.19.0-rc3 - C:\AMD\AOCL-Windows\blis/out

Where is the source code: C:/AMD/AOCL-Windows/blis

Preset: <custom>

Where to build the binaries: C:/AMD/AOCL-Windows/blis/out

Current Generator: None

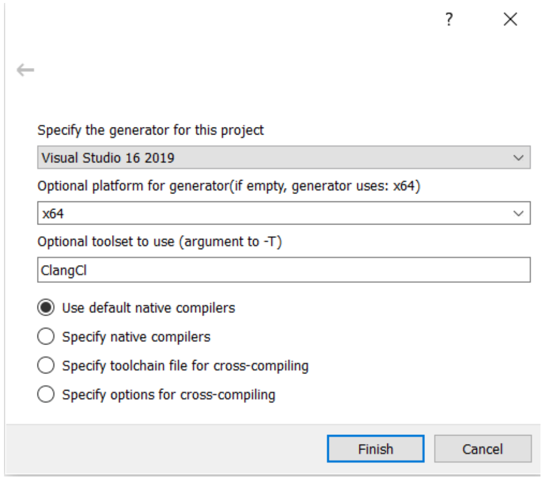
Press Configure to update and display new values in red, then press Generate to generate selected build files.

Figure 5. CMake Source and Build Folders

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

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

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



Specify the generator for this project: Visual Studio 16 2019

Optional platform for generator(if empty, generator uses: x64): x64

Optional toolset to use (argument to -T): ClangCl

[✓ Use default native compilers]
Specify native compilers
Specify toolchain file for cross-compiling
Specify options for cross-compiling

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 7. CMake Config Options

Feature	CMake Parameter
AMD CPU architecture	AOCL_BLIS_FAMILY:STRING=zen/zen2/zen3
Enable verbose mode	ENABLE_VERBOSE=ON
Shared library	BUILD_SHARED_LIBS=ON

Static library	BUILD_SHARED_LIBS=OFF
	ENABLE_AOCL_DYNAMIC=OFF

Debug/Release build type	CMAKE_BUILD_TYPE=Debug/Release
Dynamic Dispatcher	AOCL_BLIS_FAMILY:STRING=amdzen

Enable single threading	ENABLE_MULTITHREADING=OFF
	ENABLE_AOCL_DYNAMIC=OFF

Enable multi-threading with OpenMP and AOCL dynamic enabled	ENABLE_MULTITHREADING=ON
	ENABLE_OPENMP=ON
	ENABLE_AOCL_DYNAMIC=ON

Enable multi-threading with OpenMP and AOCL dynamic disabled	ENABLE_MULTITHREADING=ON
	ENABLE_OPENMP=ON
	ENABLE_AOCL_DYNAMIC=OFF

Enable BLAS/CBLAS support	ENABLE_BLAS=ON
	ENABLE_CBLAS=ON

Table 7. CMake Config Options

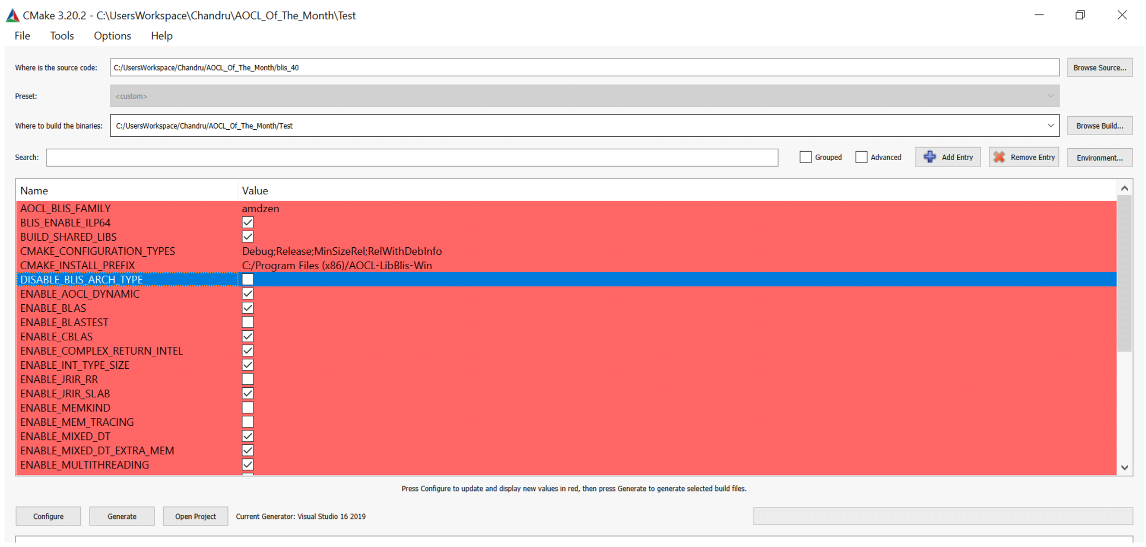
Feature	CMake Parameter

Enable 32-bit BLIS/BLAS integer size	BLIS_ENABLE_ILP64=OFF
	ENABLE_INT_TYPE_SIZE=OFF

Enable 64-bit BLIS/BLAS integer size	BLIS_ENABLE_ILP64=ON
	ENABLE_INT_TYPE_SIZE=ON

Flags that are enabled by default	ENABLE_JRIR_SLAB ENABLE_PBA_POOLS ENABLE_SBA_POOLS ENABLE_MIXED_DT ENABLE_MIXED_DT_EXTRA_MEM ENABLE_SUP_HANDLING ENABLE_PRAGMA_OMP_SIMD
Flags that are disabled by default	ENABLE_JRIR_RR ENABLE_MEM_TRACING ENABLE_MEMKIND ENABLE_SANDBOX
Use APIs without trailing underscore	ENABLE_NO_UNDERSCORE_API
Enable uppercase APIs	ENABLE_UPPERCASE_API
Absolute path to the OpenMP library, including the library name	OpenMP_libomp_LIBRARY
Disable forced code path selection using the environment variable BLIS_ARCH_TYPE	DISABLE_BLIS_ARCH_TYPE

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



Where is the source code: C:/UsersWorkspace/Chandru/AOCL_Of_The_Month/blis_40

Preset: <custom>

Where to build the binaries: C:/UsersWorkspace/Chandru/AOCL_Of_The_Month/Test

Search:

Name	Value
AOCL_BLIS_FAMILY	amdzen
BLIS_ENABLE_ILP64	✓
BUILD_SHARED_LIBS	✓
CMAKE_CONFIGURATION_TYPES	Debug;Release;MinSizeRel;RelWithDebInfo
CMAKE_INSTALL_PREFIX	C:/Program Files (x86)/AOCL-LibBlis-Win
DISABLE_BLIS_ARCH_TYPE	☐
ENABLE_AOCL_DYNAMIC	✓
ENABLE_BLAS	✓
ENABLE_BLASTEST	☐
ENABLE_CBLAS	✓
ENABLE_COMPLEX_RETURN_INTEL	✓
ENABLE_INT_TYPE_SIZE	✓
ENABLE_JRIR_RR	☐
ENABLE_JRIR_SLAB	✓
ENABLE_MEMKIND	☐
ENABLE_MEM_TRACING	☐
ENABLE_MIXED_DT	✓
ENABLE_MIXED_DT_EXTRA_MEM	✓
ENABLE_MULTITHREADING	✓

Press Configure to update and display new values in red, then press Generate to generate selected build files.

Current Generator: Visual Studio 16 2019

Figure 7. CMake Configure and Generate Project Settings

4.6.1.2 Building the Project in Visual Studio GUI

Complete the following steps in the Microsoft Visual Studio GUI:

1. Open the project generated by CMake (build folder) in "Preparing Project with CMake GUI" on page 45.
2. To generate AOCL-BLIS binaries, build the **AOCL-LibBlis-Win** project.

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

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

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

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

You can refer Table 7 and update the parameter options in the command according to the project requirements.

4.6.2.2 Building the Project in Command Prompt

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

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

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

4.6.3 Building and Running the Test Suite

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

4.6.3.1 Running Individual Tests

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

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

4.6.3.2 Running the Test Suite

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

```
Release> test_libblis.exe
```

4.6.3.3 Running Multi-thread Tests

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

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

   For example:

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

Chapter 5 AOCL-libFLAME

AOCL-libFLAME 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. libFLAME is a C-only implementation. Applications relying on standard Netlib LAPACK interfaces can utilize libFLAME with virtually no changes to their source code.

From AOCL 4.0, AMD optimized version of libFLAME(AOCL-libFLAME) is compatible with LAPACK 3.10.1 specification. In combination with the AOCL-BLIS library, which includes optimizations for the AMD “Zen”-based processors, libFLAME enables running high performing LAPACK functionalities on AMD platforms. AOCL-libFLAME supports C, FORTRAN, and C++ template interfaces (for a subset of APIs) for the LAPACK APIs.

5.1 Installing on Linux

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

5.1.1 Building AOCL-libFLAME from Source

GitHub URL: *https://github.com/amd/libflame*

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

Prerequisites

The following prerequisites must be met for installing AOCL-libFLAME:

- Target CPU ISA supporting AVX2 and FMA
- Python versions 2.7, 2.8, 3.4, and 3.6
- GNU Make 4.2
- GCC and Gfortran (versions 9.x through 11.2)

Build Steps

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

1. Clone the Git repository (*https://github.com/amd/libflame.git*).

2. Run the configure script. An example below shows therecommended 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 CXX=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>
   ```

   By default, the configuration options `--enable-amd-flags` and `--enable-amd-aocc-flags` enable multi-threading using OpenMP for the selected APIs in AOCL-libFLAME. 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
   ```

   To support binary portability across different architectures, the default compiler flags are set to `–mavx2 -mfma`. As mentioned in the Prerequisites section, AOCL-libFLAME requires target CPU to have a minimum AVX2 and FMA ISA support.

   For enabling further optimizations beyond AVX2 and FMA, you can use the configure option `--enable-optimizations` to set the desired optimization flags that will override the default flags.

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

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

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

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*.

5.1.2 Using Pre-built Libraries

You can find the AOCL-libFLAME library binaries for Linux at the following URLs:

- *https://github.com/amd/libflame/releases*
- *https://developer.amd.com/amd-aocl/blas-library/#libflame*

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

https://developer.amd.com/amd-aocl/

The tar file includes pre-built binaries of the other AMD libraries as explained in "Using Master Package" on page 15.

5.2 Usage

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

From AOCL 3.2, a separate test suite is included for the LAPACK interface. Currently, it has test cases for a few AOCL-libFLAME APIs. More test cases will be added in future releases. The test suite validates the APIs and displays performance reports. 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 the README file in the directory *test/main*.

5.2.1 Use by Applications

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

5.2.1.1 Examples

- With a static library

```
gcc test_libflame.c <path-to-libFLAME-library>/libflame.a  <path-to-BLIS-library>/libblis.a -o
test_libflame.x
```

- With a dynamic library

```
gcc test_libflame.c <path-to-libFLAME-library>/libflame.so  <path-to-BLIS-library>/libblis.so -
o test_libflame.x
```

5.3 Building AOCL-libFLAME from Source on Windows

libFLAME (*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-BLIS from Source on Windows" on page 44.

5.3.1 Building AOCL-libFLAME 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 libFLAME 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 libFLAME source tree.
2. Click on the **Configure** button to prepare the project options.
3. Set the generator to **Visual Studio 16 2019** 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 8. AOCL-libFLAME Config Options

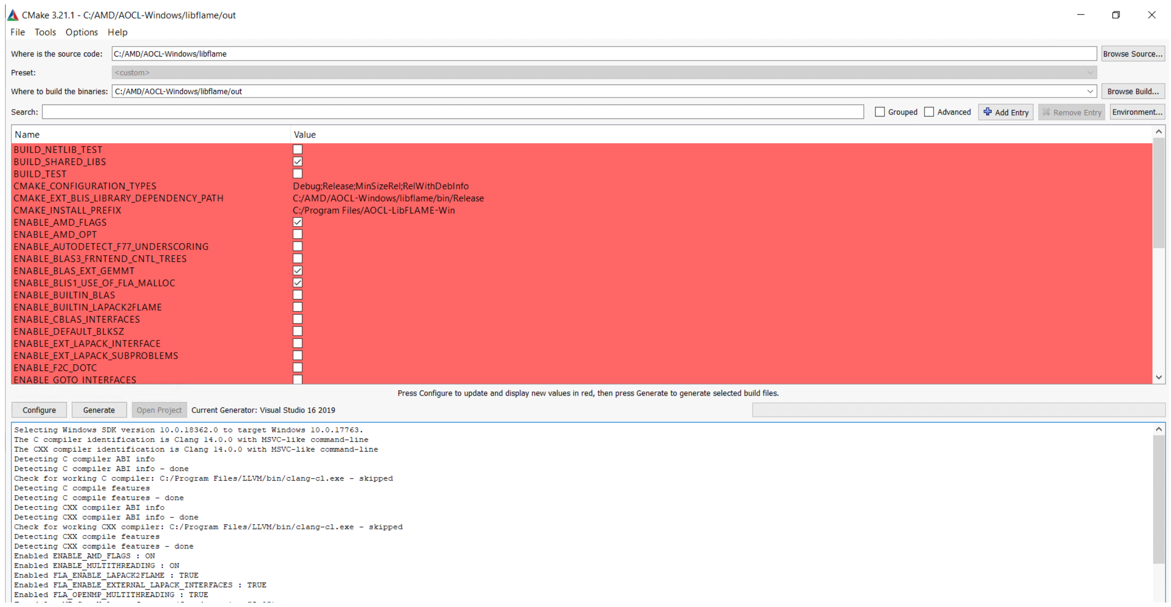
Feature	CMake Parameter(s)
Shared library	BUILD_SHARED_LIBS=ON
Static library	BUILD_SHARED_LIBS=OFF
Flags enabled by default	BUILD_SHARED_LIBS ENABLE_WINDOWS_BUILD ENABLE_AMD_FLAGS ENABLE_BLAS_EXT_GEMMT ENABLE_MULTITHREADING ENABLE_WRAPPER ENABLE_BLIS1_USE_OF_FLA_MALLOC ENABLE_BUILTIN_LAPACK2FLAME ENABLE_EXT_LAPACK_INTERFACE ENABLE_INTERNAL_ERROR_CHECKING ENABLE_NON_CRITICAL_CODE ENABLE_PORTABLE_TIMER INCLUDE_LAPACKE

Table 8. AOCL-libFLAME Config Options

Feature	CMake Parameter(s)
Enable uppercase APIs	ENABLE_UPPERCASE=ON
Enable AMD optimized path	ENABLE_AMD_OPT=ON
32-bit integer size	ENABLE_ILP64=OFF
64-bit integer size	ENABLE_ILP64=ON
AOCL-BLIS library path name	CMAKE_EXT_BLIS_LIBRARY_DEPENDENCY_PATH=<path to AOCL-BLIS library>
AOCL-BLIS library name	EXT_BLIS_LIBNAME=AOCL-BLIS library name
Enable invoking 'void' return based interface for BLAS functions DOTC and DOTU	ENABLE_F2C_DOTC=ON
Enable 'void' return type for libFLAME functions such as cladiv/ zladiv	ENABLE_VOID_RETURN_COMPLEX_FUNCTION=ON
Enables multithreading	ENABLE_MULTITHREADING=ON
Enable AMD FLAGS, internallyenables: • 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

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

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



Where is the source code: C:/AMD/AOCL-Windows/libflame

Preset: <custom>

Where to build the binaries: C:/AMD/AOCL-Windows/libflame/out

Search:

Name	Value
BUILD_NETLIB_TEST	☐
BUILD_SHARED_LIBS	✓
BUILD_TEST	☐
CMAKE_CONFIGURATION_TYPES	Debug;Release;MinSizeRel;RelWithDebInfo
CMAKE_EXT_BLIS_LIBRARY_DEPENDENCY_PATH	C:/AMD/AOCL-Windows/libflame/bin/Release
CMAKE_INSTALL_PREFIX	C:/Program Files/AOCL-LibFLAME-Win
ENABLE_AMD_FLAGS	✓
ENABLE_AMD_OPT	☐
ENABLE_AUTODETECT_F77_UNDERSCORING	☐
ENABLE_BLAS3_FRNTEND_CNTL_TREES	☐
ENABLE_BLAS_EXT_GEMMT	✓
ENABLE_BLIS1_USE_OF_FLA_MALLOC	✓
ENABLE_BUILTIN_BLAS	☐
ENABLE_BUILTIN_LAPACK2FLAME	☐
ENABLE_CBLAS_INTERFACES	☐
ENABLE_DEFAULT_BLKSZ	☐
ENABLE_EXT_LAPACK_INTERFACE	☐
ENABLE_EXT_LAPACK_SUBPROBLEMS	☐
ENABLE_F2C_DOTC	☐
ENABLE_GOTO_INTERFACES	☐

Press Configure to update and display new values in red, then press Generate to generate selected build files.

Current Generator: Visual Studio 16 2019

```
Selecting Windows SDK version 10.0.18362.0 to target Windows 10.0.17763.
The C compiler identification is Clang 14.0.0 with MSVC-like command-line
The CXX compiler identification is Clang 14.0.0 with MSVC-like command-line
Detecting C compiler ABI info
Detecting C compiler ABI info - done
Check for working C compiler: C:/Program Files/LLVM/bin/clang-cl.exe - skipped
Detecting C compile features
Detecting C compile features - done
Detecting CXX compiler ABI info
Detecting CXX compiler ABI info - done
Check for working CXX compiler: C:/Program Files/LLVM/bin/clang-cl.exe - skipped
Detecting CXX compile features
Detecting CXX compile features - done
Enabled ENABLE_AMD_FLAGS : ON
Enabled ENABLE_MULTITHREADING : ON
Enabled FLA_ENABLE_LAPACK2FLAME : TRUE
Enabled FLA_ENABLE_EXTERNAL_LAPACK_INTERFACES : TRUE
Enabled FLA_OPENMP_MULTITHREADING : TRUE
```

Figure 8. AOCL-libFLAME 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 53.
2. To generate libFLAME binaries, build the **AOCL-LibFLAME-Win** project.

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

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

5.3.2 Building AOCL-libFLAME 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 libFLAME 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 16 2019" -DCMAKE_BUILD_TYPE=Release -DBUILD_SHARED_LIBS=ON -
DEXT_BLIS_LIBNAME="AOCL-LibBlis-Win-MT-dll.lib" -
DCMAKE_EXT_BLIS_LIBRARY_DEPENDENCY_PATH="<path to AOCL-BLIS library>" -DENABLE_ILP64=ON -
DENABLE_AMD_FLAGS=ON -TLLVM -DBUILD_TEST=OFF -DBUILD_NETLIB_TEST=OFF -DENABLE_WRAPPER=ON -
DOpenMP_libomp_LIBRARY="C:\Program
Files\LLVM\lib\libomp.lib"
```

You can refer to Table 8 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.

5.4 Checking AOCL-libFLAME 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-libFLAME library.

The callback function prototype must be defined as follows:

```
int aocl_fla_progress(
char* api,
integer lenapi,
integer *progress,
integer *current_thread,
integer *total_threads
)
```

However, the function name can be changed as per your preference.

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

Table 9. AOCL-libFLAME Progress Feature Callback Function Parameters

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

Callback Registration

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

```
aocl_fla_set_progress(test_progress);   // for AOCL-libFLAME
```

Example:

```
int aocl_fla_progress(char* api,integer lenapi,integer *progress,integer
*current_thread,integer *total_threads)
{
  char buf[BUFLEN];
  if( lenapi >= BUFLEN ) lenapi = BUFLEN-1;
  strncpy( buf, api, lenapi );
  buf[lenapi] = '\0';
  printf( "In AOCL FLA  Progress thread  %lld", at API  %s, progress  %lld total threads=
%lld\n", *current_thread, buf, *progress,*total_threads );
  return 0;

}

or

int test_progress(char* api,integer lenapi,integer *progress,integer *current_thread,integer
*total_threads)
{
  char buf[BUFLEN];
  if( lenapi >= BUFLEN ) lenapi = BUFLEN-1;
  strncpy( buf, api, lenapi );
  buf[lenapi] = '\0';
  printf( "In AOCL Progress thread  %lld", at API  %s, progress  %lld total threads= %lld\n",
*current_thread, buf, *progress,*total_threads );
  return 0;

}

Register the callback with:
aocl_fla_set_progress(test_progress);
```

Limitations

On Windows, aocl_fla_progress is not supported when using AOCL-libFLAME. 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^{TM} 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^{TM} 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^{TM} 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 --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 --
prefix=<your-install-dir>

Long double FFTW libraries

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

Quad Precision FFTW libraries

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

Quad FFTW libraries

$ ./configure --enable-shared --enable-openmp --enable-quad-precision --enable-amd-opt
--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^{TM} 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 130.

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. Presently, it is supported for the GCC compiler and 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 10. 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 10. AOCL-FFTW Config Options

Feature	CMake Parameters

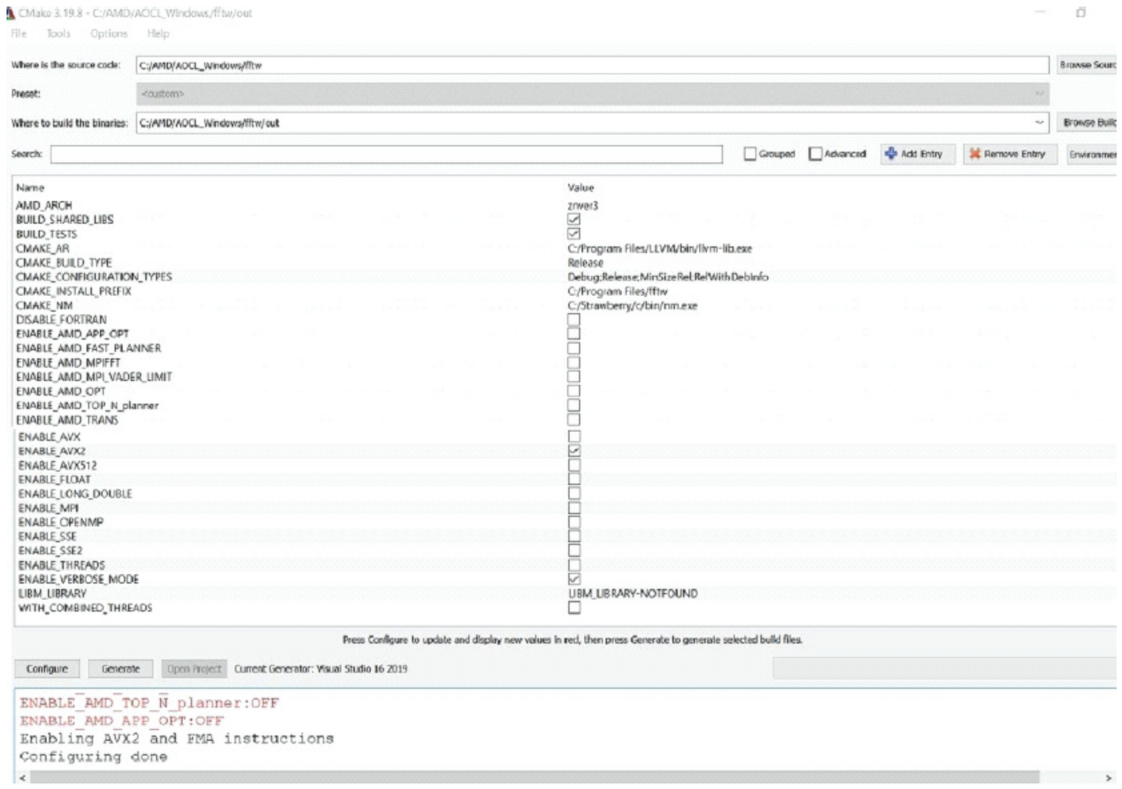
Flags for enhanced instruction set support	ENABLE_SSE=ON
	ENABLE_SSE2=ON
	ENABLE_AVX=ON
	ENABLE_AVX2=ON
	ENABLE_AVX512=ON

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:



CMake 3.19.8 - C:/AMD/AOCL_Windows/fftw/out

Where is the source code: C:/AMD/AOCL_Windows/fftw

Preset: <custom>

Where to build the binaries: C:/AMD/AOCL_Windows/fftw/out

Name	Value
AMD_ARCH	znver3
BUILD_SHARED_LIBS	✓
BUILD_TESTS	✓
CMAKE_AR	C:/Program Files/LLVM/bin/llvm-lib.exe
CMAKE_BUILD_TYPE	Release
CMAKE_CONFIGURATION_TYPES	Debug;Release;MinSizeRel;RelWithDebInfo
CMAKE_INSTALL_PREFIX	C:/Program Files/fftw
CMAKE_NM	C:/Strawberry/c/bin/nm.exe
DISABLE_FORTRAN	☐
ENABLE_AMD_APP_OPT	☐
ENABLE_AMD_FAST_PLANNER	☐
ENABLE_AMD_MPIFFT	☐
ENABLE_AMD_MPI_VADER_LIMIT	☐
ENABLE_AMD_OPT	☐
ENABLE_AMD_TOP_N_planner	☐
ENABLE_AMD_TRANS	☐
ENABLE_AVX	☐
ENABLE_AVX2	✓
ENABLE_AVX512	☐
ENABLE_FLOAT	☐
ENABLE_LONG_DOUBLE	☐
ENABLE_MPI	☐
ENABLE_OPENMP	☐
ENABLE_SSE	☐
ENABLE_SSE2	☐
ENABLE_THREADS	☐
ENABLE_VERBOSE_MODE	✓
LIBM_LIBRARY	LIBM_LIBRARY-NOTFOUND
WITH_COMBINED_THREADS	☐

Press Configure to update and display new values in red, then press Generate to generate selected build files.

Current Generator: Visual Studio 16 2019

```
ENABLE_AMD_TOP_N_planner:OFF
ENABLE_AMD_APP_OPT:OFF
Enabling AVX2 and FMA instructions
Configuring done
```

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 10 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://developer.amd.com/amd-aocl/fftw/

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://developer.amd.com/amd-aocl/

The *tar* and *zip* files include pre-built binaries of other AMD libraries as explained in "Using Master Package" on page 15.

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 130.

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 130.

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 130.

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 software library containing a collection of basic math functions optimized for x86-64 processor-based machines. It provides many routines from the list of standard C99 math functions. It includes scalar and vector variants of the core math functions. AOCL-LibM is a C library you can link to your applications to replace the compiler provided math functions. After linking, the applications can invoke math functions instead of compiler math functions for better accuracy and performance.

The latest AOCL-LibM includes the alpha version of the vector variants for the core math functions; power, exponential, logarithmic, and trigonometric. A few caveats of the vector variants are as follows:

- The vector variants are the relaxed versions of the respective math functions with respect to the accuracy.
- The routines take advantage of the AMD64 architecture for the performance. Some of the performance is gained by sacrificing error handling or the acceptance of certain arguments.
- Abnormal inputs may produce unpredictable results. It is therefore the responsibility of the caller of these routines to ensure that their arguments are suitable.
- The vector variants are not expected to set the IEEE error codes, it is recommended not to rely on error codes for the vector variants.
- The vector routines must be invoked using the C intrinsics or from the x86 assembly.

The vector variants can be enabled by using AOCC compiler with `-ffast-math` flag and it is not recommended to call these functions manually. As these functions accept arguments in __m128, __m128d, __m256, __m256d,__m512 and __m512d types, you must manually pack-unpack to/from such a format.

However, the symbols are enabled in library and the signatures use the naming convention as follows:

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

Where,

- v – vector
- r – real
- a – array
- <type> - ‘s’ for single precision and ‘d’ for double precision
- <vec_size> - 2, 4, 8, or 16 for 2, 4, 8, or 16 element vector respectively
- <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 the functions have an 'amd_' prefix and it is omitted from the list to reduce the length.*

- 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, and vrd4_tan
  - 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, amd_vrd8_sin, and vrd8_tan
- Inverse Trigonometric
  - vrs4_acosf, vrs4_asinf, and vrs8_asinf
  - vrs4_atanf, vrs8_atanf, vrd2_atan
  - vrs16_atanf and vrs16_asinf
  - vrd8_atan and vrd8_asin

- Hyperbolic
  - vrs4_coshf and vrs4_tanhf
  - vrs8_coshf and vrs8_tanhf
- Power
  - vrs4_powf, vrd2_pow, vrd4_pow, vrs8_powf, and vrsa_powf
  - vrs16_powf and vrd8_pow
- Error

  vrs8_erff and vrs4_erff

The following scalar functions are present in the library:

They can be called by a standard C99 function call and naming convention and must be linked with AOCL-LibM before standard `‘libm`.

For example:

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

Or

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

The following functions have vector variants in AOCL-LibM:

- Trigonometric

  cosf, cos, sinf, sin, tanf, tan, sincosf, and sincos
- Inverse Trigonometric

  acosf, acos, asinf, asin, atanf, atan, atan2f, and atan2
- Hyperbolic

  coshf, cosh, sinhf, sinh, tanhf, and tanh
- Inverse Hyperbolic

  acoshf, acosh, asinhf, asinh, atanhf, and atanh
- Exponential and Logarithmic
  - expf, exp, exp2f, exp2, exp10f, exp10, expm1f, and expm1
  - logf, log, log10f, log10, log2f, log2, log1pf, and log1p
  - logbf, logb, ilogbf, and ilogb
  - modff, modf, frexpf, frexp, ldexpf, and ldexp
  - scalbnf, scalbn, scalblnf, and scalbln
- Error

  erff

- Power and Absolute Value
  - powf, pow, fastpow, cbrtf, cbrt, sqrtf, sqrt, hypotf, and hypot
  - fabsf and fabs
- Nearest Integer
  - ceilf, ceil, floorf, floor, truncf, and trunc
  - rintf, rint, roundf, round, nearbyintf, and nearbyint
  - lrintf, lrint, llrintf, and llrint
  - lroundf, lround, llroundf, and llround
- Remainder

  fmodf, fmod, remainderf, and remainder
- Manipulation
  - copysignf, copysign, nanf, nan, finitef, and finite
  - nextafterf, nextafter, nexttowardf, and nexttoward
- Maximum, Minimum, and Difference

  fdimf, fdim, fmaxf, fmax, fminf, and fmin

A fast version of AOCL LibM is released as a part of the package libalmfast.so. This library contains faster variants of some of the scalar functions. These function calls are enabled by the compiler based on certain flags used. For more information, refer to the AOCC 4.0 user guide.

7.1 Installation on Linux

AOCL-LibM binary for Linux isavailable at the following URL:

https://developer.amd.com/amd-aocl/amd-math-library-libm/

Also, LibM binary can be installed from the AOCC and GCC compiled AOCL master installer tar file available on AMD Developer Central (*https://developer.amd.com/amd-aocl/#download*).

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

7.2 Compiling AOCL-LibM

Minimum software requirements for compilation:

- GCC versions 9.3.0 through v11.2.0
- Glibc versions 2.29 through v2.31
- Clang 12.0.0 (AOCC 3.0) through Clang 14.0.0 (AOCC 4.0)
- Virtualenv with Python 3.6.8
- SCons versions 3.0.5 through 4.2.0

Complete the following steps to compile AOCL-LibM:

1. Download source from GitHub (*https://github.com/amd/aocl-libm-ose*).
2. Navigate to the LibM folder and checkout to the branch *aocl-4.0*:

   ```
   cd aocl-libm-ose
   git checkout aocl-4.0
   ```

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:

   ```
   scons -j32

   Additional parameters: install --prefix=<path to install> ALM_CC=<gcc/clang exe path>

   Verbosity options: --verbose=1

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

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

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

7.3 Usage

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

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

The Linux libraries may have a dependency on the system math library. When linking AOCL-LibM, ensure that it precedes the system math library in the link order that is, `-lalm` must appear before `-lm`. The explicit linking of the system math library is required when using the GCC/AOCC compiler. 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 AMD LibM scalar functions, use the following commands:

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

For the vector calls, you must depend on compiler flag `-ffast-math`.

Though not recommended, you can call the functions directly with manual packing and unpacking. To invoke the vector functions directly, you must include the header file *amdlibm_vec.h*. The following program shows such an example with both returning and storing the values in an array. For simplicity, the size and other checks are omitted from the example.

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

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(ip1[3], ip1[2], ip1[1], ip1[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/AMD LibM
$ clang -Wall -std=c99 -ffastmath myprogram.c -o myprogram –L<path to libalm.so> -lalm -lm
```

7.4 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 python3
- SCons 4.4.0

Complete the following steps to compile AOCL-LibM on Windows:

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

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

3. Install virtualenv:

   ```
   pip install virtualenv
   ```

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

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

5. Activate virtual environment and install SCons inside:

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

6. Build the project using clang compiler:

   ```
   scons -j32 ALM_CC=<clang-cl executable path> ALM_CXX=<clang-cl executable path>
   Verbosity options: --verbose=1
   Debug mode build: --debug_mode=all
   ```

   For example:

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

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

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

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-BLIS and AOCL-libFLAME is recommended.

8.1 Installation

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

8.1.1 Building AOCL-ScaLAPACK from Source

Note: *Starting from AOCL 3.1, the AOCL-ScaLAPACK will be available in the new GitHub repository (https://github.com/amd/aocl-scalapack). The older GitHub repository (https://github.com/amd/scalapack) is deprecated.*

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-BLIS
- AOCL-libFLAME
- An MPI library (validated with OpenMPI library)

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

1. Clone the GitHub repository (*https://github.com/amd/aocl-scalapack.git*).
2. Execute the command:
   ```
   $ cd scalapack
   ```
3. CMake as follows:
   a. Create a new directory. For example, build:
      ```
      $ mkdir build
      $ cd build
      ```
   b. Set PATH and LD_LIBRARY_PATH appropriately to the MPI installation.

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

Table 11. Compiler and Type of Library

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

Table 11. Compiler and Type of Library

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

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-BLIS library>/
libblis-mt.a" -DLAPACK_LIBRARIES="<path to libflame library>/libflame.a"   -
DBLACS_LIBRARIES=<path to BLACS library>/libBLACS.a  -DCMAKE_C_COMPILER=mpicc -
DCMAKE_Fortran_COMPILER=mpif90 -DUSE_OPTIMIZED_LAPACK_BLAS=OFF
```

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

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

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

d. Ensure CMake locates AOCL-libFLAME and AOCL-BLIS 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 URLs:

- *https://github.com/amd/aocl-scalapack/releases*
- *https://developer.amd.com/amd-aocl/scalapack/*

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

https://developer.amd.com/amd-aocl/

The tar file includes pre-built binaries of other AMD Libraries as explained in "Using Master Package" on page 12.

8.2 Usage

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

```
$ cd scalapack/TESTING
```

8.3 Building AOCL-ScaLAPACK from Source on Windows

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

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

Prerequisites

The following prerequisites must be met:

- Windows10/11 or Windows Server 2019/2022
- LLVM 13/14 for AMD “Zen3” and AMD “Zen4” support (or LLVM 11 for AMD “Zen2” support)
- LLVM plug-in for Microsoft Visual Studio (if latest version of LLVM is installed separately, this plug-in enables linking Microsoft Visual Studio with the installed LLVM tool-chain)
- CMake versions 3.0 through 3.23.3
- MPI compiler
- Fortran 90 compiler
- Microsoft Visual Studio 2019 (build 16.8.7) through 2022 (build 17.3.2)
- Microsoft Visual Studio tools
  - Python development
  - Desktop development with C++: C++ Clang-Cl for v142 build tool (x64 or x86)

8.3.1 Building AOCL-ScaLAPACK Using GUI

8.3.1.1 Preparing Project with CMake GUI

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

1. Set the **source** (folder containing aocl-scalapack source code) and **build** (folder in which the project files will be generated, for example, **out**) folder paths. It is not recommended to use the folder named **build** as a folder with that name exists at the top of libFLAME source tree.
2. Click on the **Configure** button to prepare the project options.
3. Set the generator to **Visual Studio 16 2019** and the compiler to **ClangCl** or **LLVM**.
4. Update the options based on the project requirements. All the available options are listed in the following table:

Table 12. AOCL-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`

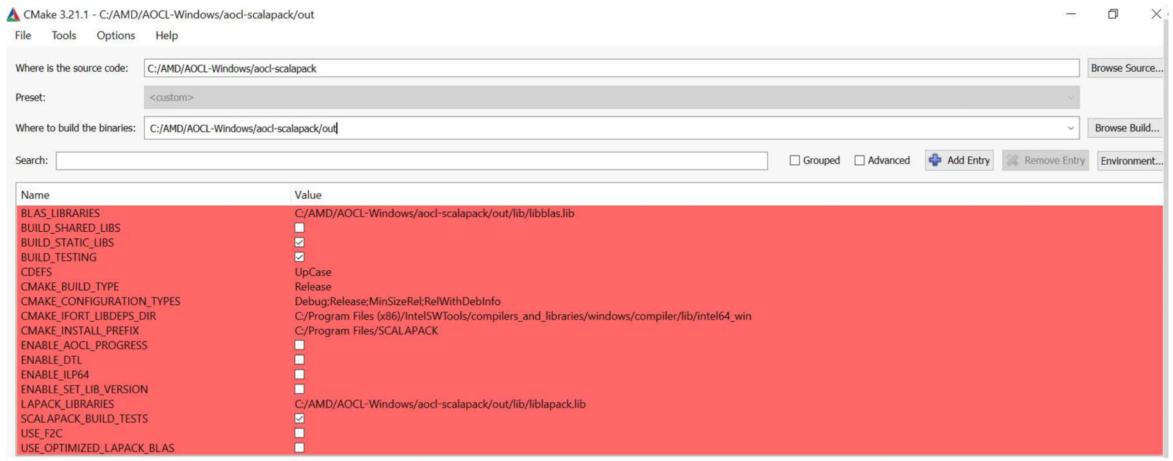
Static library	`BUILD_STATIC_LIBS=ON`
	`BUILD_SHARED_LIBS=OFF`

Provide external BLAS/BLIS library	`BLAS_LIBRARIES`
	`=<Path to BLAS/BLIS lib>`

Table 12. AOCL-ScaLAPACK CMake Parameter List

Provide external LAPACK/libFLAME library	`LAPACK_LIBRARIES =<Path to lapack/libflame lib>`
Integer bit length: • ON => 64-bit integer length • OFF => 32-bit integer length	`ENABLE_ILP64`
Flags disabled by default	`USE_OPTIMIZED_LAPACK_BLAS`

5. Select the available and recommended options as follows:



Where is the source code: C:/AMD/AOCL-Windows/aocl-scalapack

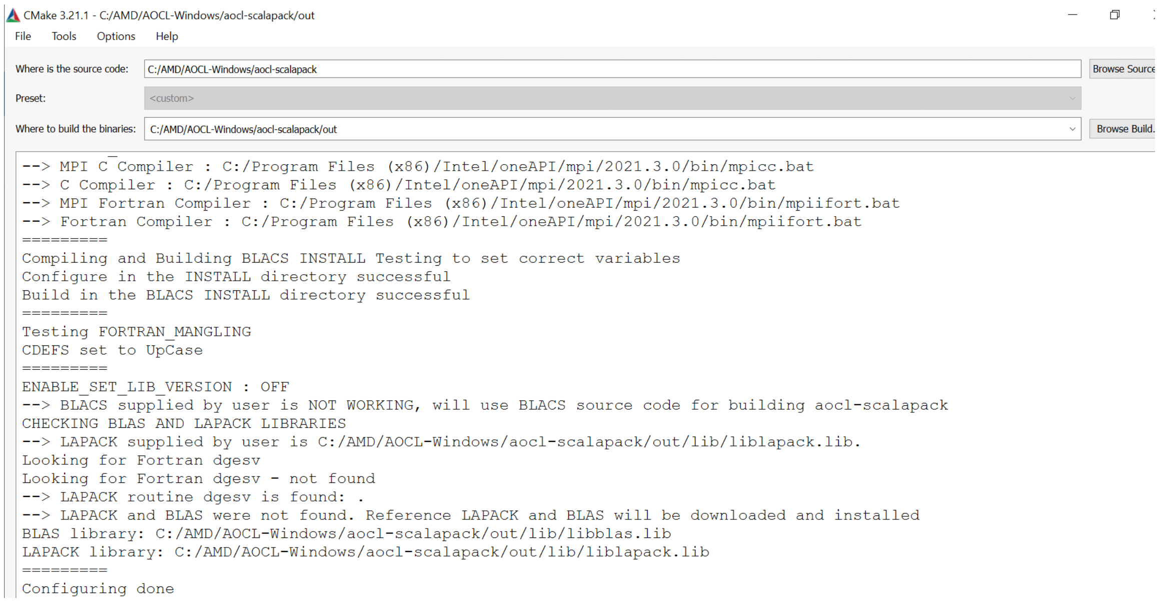
Preset: <custom>

Where to build the binaries: C:/AMD/AOCL-Windows/aocl-scalapack/out

Search:

Name	Value
BLAS_LIBRARIES	C:/AMD/AOCL-Windows/aocl-scalapack/out/lib/libblas.lib
BUILD_SHARED_LIBS	☐
BUILD_STATIC_LIBS	✓
BUILD_TESTING	✓
CDEFS	UpCase
CMAKE_BUILD_TYPE	Release
CMAKE_CONFIGURATION_TYPES	Debug;Release;MinSizeRel;RelWithDebInfo
CMAKE_IFORT_LIBDEPS_DIR	C:/Program Files (x86)/IntelSWTools/compilers_and_libraries/windows/compiler/lib/intel64_win
CMAKE_INSTALL_PREFIX	C:/Program Files/SCALAPACK
ENABLE_AOCL_PROGRESS	☐
ENABLE_DTL	☐
ENABLE_ILP64	☐
ENABLE_SET_LIB_VERSION	☐
LAPACK_LIBRARIES	C:/AMD/AOCL-Windows/aocl-scalapack/out/lib/liblapack.lib
SCALAPACK_BUILD_TESTS	✓
USE_F2C	☐
USE_OPTIMIZED_LAPACK_BLAS	☐

Figure 10. AOCL-ScaLAPACK CMake Options



Where is the source code: C:/AMD/AOCL-Windows/aocl-scalapack

Preset: <custom>

Where to build the binaries: C:/AMD/AOCL-Windows/aocl-scalapack/out

```
--> MPI C Compiler : C:/Program Files (x86)/Intel/oneAPI/mpi/2021.3.0/bin/mpicc.bat
--> C Compiler : C:/Program Files (x86)/Intel/oneAPI/mpi/2021.3.0/bin/mpicc.bat
--> MPI Fortran Compiler : C:/Program Files (x86)/Intel/oneAPI/mpi/2021.3.0/bin/mpiifort.bat
--> Fortran Compiler : C:/Program Files (x86)/Intel/oneAPI/mpi/2021.3.0/bin/mpiifort.bat
=========
Compiling and Building BLACS INSTALL Testing to set correct variables
Configure in the INSTALL directory successful
Build in the BLACS INSTALL directory successful
=========
Testing FORTRAN_MANGLING
CDEFS set to UpCase
=========
ENABLE_SET_LIB_VERSION : OFF
--> BLACS supplied by user is NOT WORKING, will use BLACS source code for building aocl-scalapack
CHECKING BLAS AND LAPACK LIBRARIES
--> LAPACK supplied by user is C:/AMD/AOCL-Windows/aocl-scalapack/out/lib/liblapack.lib.
Looking for Fortran dgesv
Looking for Fortran dgesv - not found
--> LAPACK routine dgesv is found: .
--> LAPACK and BLAS were not found. Reference LAPACK and BLAS will be downloaded and installed
BLAS library: C:/AMD/AOCL-Windows/aocl-scalapack/out/lib/libblas.lib
LAPACK library: C:/AMD/AOCL-Windows/aocl-scalapack/out/lib/liblapack.lib
=========
Configuring done
```

Figure 11. AOCL-ScaLAPACK CMake Config Options

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

8.3.1.2 Building the Project in Visual Studio GUI

Complete the following steps in Microsoft Visual Studio GUI:

1. Open the project generated by CMake (build folder) in “Preparing Project with CMake GUI” on page 78.
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 16 2019" -DCMAKE_BUILD_TYPE=Release
-DBUILD_SHARED_LIBS=ON
-DBUILD_STATIC_LIBS=OFF -DBLAS_LIBRARIES="<path to BLIS library>/AOCL-
LibBlis-Win-MT-dll.lib"
-DLAPACK_LIBRARIES="<path to libflame library>/AOCL-LibFLAME-Win-MT-dll.lib"
```

Refer Table 12 to update the parameter options in the command according to the project requirements.

8.3.2.2 Building the Project in Command Prompt

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

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

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

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

8.3.2.3 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.

8.3.2.4 Running Individual Tests

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

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

Table 13. 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_scalapack_progress(char* api, int *lenapi, int *progress, int *mpi_rank, 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.

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 and Mersenne Twister. The library contains six base generators and twenty-three distribution generators. In addition, you can supply a custom-built generator as the base generator for all the distribution generators.

9.1 Installation

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

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

https://developer.amd.com/amd-aocl/rng-library/

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

https://developer.amd.com/amd-aocl/

- The tar file includes pre-built binaries of other AMD libraries as explained in Using Master Package.

To install the AOCL-RNG binary for Windows, refer to section 3.2.4. *rng_amd.dll* and *rng_amd.lib* are a part of the dynamic library and *rng_amd-static.lib* is a static library.

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 := -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 16 2019.
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 same project folder.
6. Use the following navigation to add WIN64 preprocessor definition:

   Project >**Properties** >**C/C++** >**Preprocessor** >**Preprocessor Definitions**
7. Compile and then run the application.
8. You may create Fortran based project in similar manner and compile it using ifort compiler.
9. You can also compile your application using AOCL-RNG static library (*rng_amd-static.lib*).

Chapter 10 AOCL-SecureRNG

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

10.1 Installation

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

https://developer.amd.com/amd-aocl/rng-library/

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

https://developer.amd.com/amd-aocl/

- The tar file includes pre-built binaries of other AMD libraries as explained in “Using Master Package” on page 15.

To install the AOCL-SecureRNG binary for Windows, refer to section 3.2.4. *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 16 2019.
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 RNG APIs under window application.
5. Copy the AOCL-SecureRNG header file (*secrng.h*) and AOCL-RNG 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 the sparse matrices and vectors. It is designed to be used with C and C++.

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

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

The list of supported functions is as follows:

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

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

Notes:

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

Multi-thread Support

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

- aoclsparse_xcsrmv (single and double precision)
- aoclsparse_xellmv (single and double precision)
- aoclsparse_dmv (double precision)

For more information on performing multi-thread runs, refer “Simple Test” on page 93.

For more information on the AOCL-Sparse APIs, refer *aocl-sparse_API_Guide.pdf* in the source directory (*https://github.com/amd/aocl-sparse*).

11.1 Installation

11.1.1 Building AOCL-Sparse from Source on Linux

The following prerequisites must be met:

- Git
- CMake versions 3.5 through 3.19.6
- Boost library versions 1.65 through 1.77

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

1. Download the latest release of aocl-sparse (*https://github.com/amd/aocl-sparse*).
2. Clone the Git repository (*https://github.com/amd/aocl-sparse.git*).
3. Run the command:

   ```
   cd aocl-sparse
   ```

4. Create the build directory and change to it:

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

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

Table 14. Compiler and Library Type

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

6. From AOCL-Sparse 4.0, AVX-512 kernels for SPMV are supported as an experimental feature. It can be enabled using the CMake option:

```
-DUSE_AVX512
Example: cmake ../.. -DUSE_AVX512=1 -DBUILD_SHARED_LIBS=OFF
```

Note: *Ensure that the target CPU supports AVX-512 ISA.*

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

Table 15. AOCL-Sparse - CMake Build Options

Build Option	Feature
CMAKE_INSTALL_PREFIX	Use -DCMAKE_INSTALL_PREFIX=<path> to choose the custom path. The default install path is */opt/aoclsparse/*
CMAKE_BUILD_TYPE	• release => Release Library (Default) • debug => Debug Library
CMAKE_CXX_COMPILER	Use -DCMAKE_CXX_COMPILER=clang++ for AOCC builds
BUILD_SHARED_LIBS	• OFF => Build Static Library • ON => Build Dynamic/Shared library (Default)
BUILD_ILP64	Integer length: • OFF => 32-bit integer length (Default) • ON => 64-bit integer length
SUPPORT_OMP	Multi-threading using OpenMP: • OFF => Disable OpenMP • ON => Enable OpenMP (Default)
COMPILER_FLAGS	If defined, requires a valid compiler flag: • not defined => Use default flags • -march=znver1 => specific for AMD "Zen1" processors • -march=znver2 => specific for AMD "Zen2" and compatible with AMD "Zen1" processors • -march=znver3 => specific for AMD "Zen3" and compatible with AMD "Zen1"/AMD "Zen2" processors • -march=znver4 => specific for AMD "Zen4" and compatible with AMD "Zen1"/AMD "Zen2"/AMD "Zen3" processors
USE_AVX512	• OFF => compiler flags depend upon the build flag COMPILER_FLAGS (Default) • ON => Enable AVX512 kernels for SPMV
BUILD_CLIENTS_BENCHMARKS	• OFF => Disable building benchmarks (Default) • ON => Build client benchmarking (requires Boost library)
BUILD_CLIENTS_SAMPLES	• OFF => disable building sparse API examples (Default) • ON => enable building sparse examples for SPMV, CSR2M, DTRSV, CG, and GMRES
BUILD_CLIENTS_TESTS	• OFF => Disable building functionality/unit tests (Default) • ON => Enable building functionality/unit tests for Hint, TRSV, and CG

8. Build the aocl-sparse library:

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

9. Install aocl-sparse to the directory */opt/aoclsparse* or a custom path:

```
$ make install
```

11.1.2 Simple Test

After compiling the library with benchmarks, run the following aocl-sparse example to test the installation:

1. Navigate to the test binary directory:

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

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

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

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

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

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

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

11.1.3 Using Pre-built Libraries

You can find the AMD optimized AOCL-Sparse library binaries for Linux at the following URLs:

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

https://developer.amd.com/amd-aocl/aocl-sparse/

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

https://developer.amd.com/amd-aocl/

The tar file includes pre-built binaries of other AMD libraries as explained in “Using Master Package” on page 15.

11.2 Usage on Linux

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

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

The sample programs are built as a part of AOCL-Sparse's CMake build system by enabling the flag BUILD_CLIENTS_SAMPLES and the binaries are located in *<build_directory>/tests/examples*.

11.2.1 Use by Applications

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

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

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

Examples to Build Sample Applications

```
SPMV:
g++ sample_spmv.cpp -I<path-to-aocl-sparse-header> -L<path-to-aocl-sparse-library>/
-laoclsparse -o test_aoclsparse.x
```

```
CSR2M:
g++ sample_csr2m.cpp -I<path-to-aocl-sparse-header> -L<path-to-aocl-sparse-library>/
-laoclsparse -o test_aoclsparse.x
```

```
TRSV:
g++ sample_dtrsv.cpp -I<path-to-aocl-sparse-header> -L<path-to-aocl-sparse-library>/
-laoclsparse -o test_aoclsparse.x

CG example using Reverse Communication Interface(RCI):
g++ sample_itsol_d_cg_rci.cpp -I<path-to-aocl-sparse-header> -L<path-to-aocl-sparse-library>/
-laoclsparse  -o test_aoclsparse.x

CG example using Direct Interface:
g++ sample_itsol_d_cg.cpp -I<path-to-aocl-sparse-header> -L<path-to-aocl-sparse-library>/
-laoclsparse  -o test_aoclsparse.x

GMRES example using Reverse Communication Interface(RCI):
g++ sample_itsol_d_gmres_rci.cpp -I<path-to-aocl-sparse-header> -L<path-to-aocl-sparse-
library>/
-laoclsparse  -o test_aoclsparse.x

GMRES example using Direct Interface:
g++ sample_itsol_d_gmres.cpp -I<path-to-aocl-sparse-header> -L<path-to-aocl-sparse-library>/
-laoclsparse  -o test_aoclsparse.x
```

The following is a sample *cpp* file depicting the AOCL-Sparse spmv API usage:

```
//file :sample_spmv.cpp
#include "aoclsparse.h"
#include <iostream>

#define M 5
#define N 5
#define NNZ 8

int main(int argc, char* argv[])
{
    aoclsparse_operation   trans     = aoclsparse_operation_none;

    double alpha = 1.0;
    double beta  = 0.0;

    // Print aoclsparse version
    std::cout << aoclsparse_get_version() << std::endl;

    // Create matrix descriptor
    aoclsparse_mat_descr descr;
    // aoclsparse_create_mat_descr set aoclsparse_matrix_type to aoclsparse_matrix_type_general
    // and aoclsparse_index_base to aoclsparse_index_base_zero.
    aoclsparse_create_mat_descr(&descr);

aoclsparse_index_base base = aoclsparse_index_base_zero;

    // Initialise matrix
    //  1  0  0  2  0
    //  0  3  0  0  0
    //  0  0  4  0  0
    //  0  5  0  6  7
    //  0  0  0  0  8
    aoclsparse_int csr_row_ptr[M+1] = {0, 2, 3, 4, 7, 8};
    aoclsparse_int csr_col_ind[NNZ]= {0, 3, 1, 2, 1, 3, 4, 4};
    double         csr_val[NNZ] = {1, 2, 3, 4, 5, 6, 7, 8};
    aoclsparse_matrix A;
    aoclsparse_create_dcsr(A, base, M, N, NNZ, csr_row_ptr, csr_col_ind, csr_val);

    // Initialise vectors
    double x[N] = { 1.0, 2.0, 3.0, 4.0, 5.0};
    double y[M];

    //to identify hint id(which routine is to be executed, destroyed later)
    aoclsparse_set_mv_hint(A, trans, descr, 1);
```

```
 // Optimize the matrix, "A"
    aoclsparse_optimize(A);

    std::cout << "Invoking aoclsparse_dmv..";
    //Invoke SPMV API (double precision)
    aoclsparse_dmv(trans,
    &alpha,
    A,
    descr,
    x,
    &beta,
    y);
    std::cout << "Done." << std::endl;
    std::cout << "Output Vector:" << std::endl;
    for(aoclsparse_int i=0;i < M; i++)
std::cout << y[i] << std::endl;

    aoclsparse_destroy_mat_descr(descr);
    aoclsparse_destroy(A);
    return 0;
}
```

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

```
g++ sample_csrmv.cpp -I<path-to-aocl-sparse-header> -L<path-to aocl-sparse-library> -
laoclsparse -o test_aoclsparse.x
```

11.3 Build AOCL-Sparse from Source on Windows

GitHub URL: *https://github.com/amd/aocl-sparse*

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

Prerequisites

For more information, refer to the Prerequisites sub-section in section “Build AOCL-BLIS from Source on Windows” on page 44.

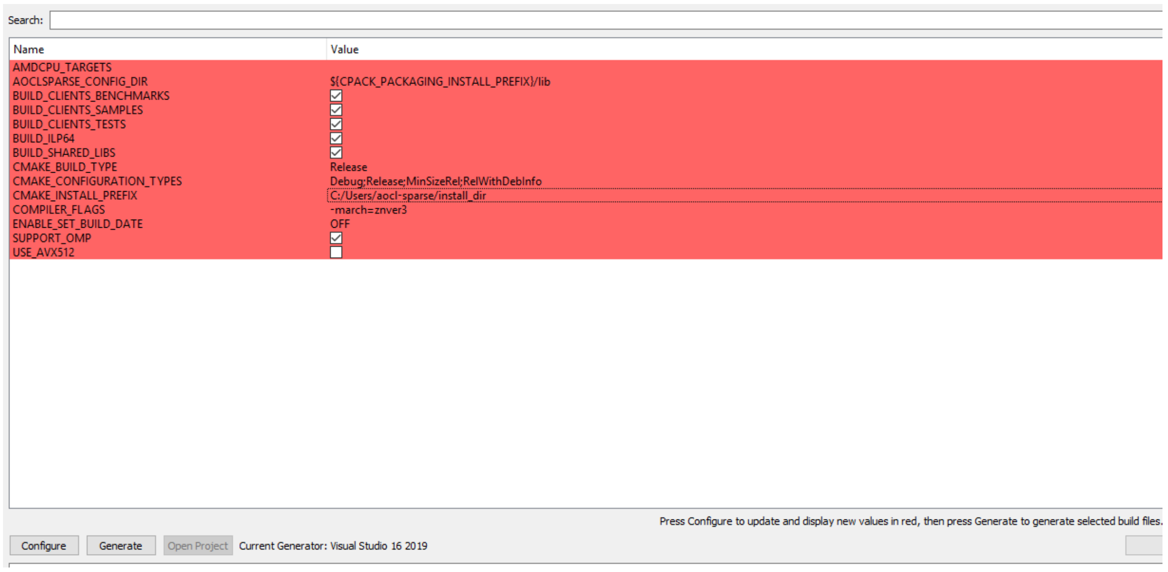
11.3.1 Building AOCL-Sparse Using GUI

11.3.1.1 Preparing Project with CMake GUI

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

1. Set the **source** (folder containing aocl-sparse 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 it is already used for Linux build system.
2. Click on the **Configure** button to prepare the project options.

3. Set the generator to **Visual Studio 16 2019** and the compiler to **ClangCl**.
4. Update the options based on the project requirements. All the available options are listed in Table 15.Select the available and recommended options as follows:



Name	Value
AMDCPU_TARGETS	
AOCLSPARSE_CONFIG_DIR	${CPACK_PACKAGING_INSTALL_PREFIX}/lib
BUILD_CLIENTS_BENCHMARKS	✓
BUILD_CLIENTS_SAMPLES	✓
BUILD_CLIENTS_TESTS	✓
BUILD_ILP64	✓
BUILD_SHARED_LIBS	✓
CMAKE_BUILD_TYPE	Release
CMAKE_CONFIGURATION_TYPES	Debug;Release;MinSizeRel;RelWithDebInfo
CMAKE_INSTALL_PREFIX	C:/Users/aocl-sparse/install_dir
COMPILER_FLAGS	-march=znver3
ENABLE_SET_BUILD_DATE	OFF
SUPPORT_OMP	✓
USE_AVX512	☐

Press Configure to update and display new values in red, then press Generate to generate selected build files.

Current Generator: Visual Studio 16 2019

Figure 12. AOCL-Sparse CMake Config Options

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

11.3.1.2 Building the Project in Visual Studio GUI

Complete the following steps in Microsoft Visual Studio GUI:

1. Open the project generated by CMake (build folder) in "Preparing Project with CMake GUI" on page 96.
2. To generate the AOCL-Sparse binaries, build the **AOCL-Sparse** project. The library files would be generated in the folder **bin** based on the project settings.

For example:

aocl-sparse/build/library/Release/aoclsparse.dll

aoclsparse.lib

11.3.2 Building AOCL-Sparse using Command-line Arguments

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

11.3.2.1 Configuring the Project in Command Prompt

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

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

   ```
   cmake .. -DBUILD_ILP64=OFF -DBUILD_SHARED_LIBS=ON -DBUILD_CLIENTS_BENCHMARKS=ON -
   DBUILD_CLIENTS_SAMPLES=ON -G "Visual
   Studio 16 2019" -T LLVM
   ```

   Refer Table 15 to update the parameter options in the command according to the project requirements.

11.3.2.2 Building the Project in Command Prompt

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

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

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

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

11.3.2.3 Building and Running the Test Suite

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

11.3.2.4 Running Individual Tests

The AOCL-Sparse executable accepts 2 types of inputs, namely randomly generated matrix data and matrices in Matrix Market format (mtx). The MTX inputs can be downloaded from SuiteSparse Matrix Collection website (*https://sparse.tamu.edu/*). Usage of both the type of inputs is shown below.

Copy the generated library and test bench from the release folder to *<aocl-sparse>/tests/staging/ Release*. Run the tests from the command prompt as follows:

```

Random Data:
./aoclsparse-bench.exe --function=optmv --precision=d --sizem=1000 --sizen=1000 --sizennz=4000
-verify=1

MTX Input:
./aoclsparse-bench.exe --function=optmv --precision=d --mtx=LFAT5.mtx --verify=1
```

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

```
set AOCLSPARSE_NUM_THREADS=4
./aoclsparse-bench.exe --function=csrmv --precision=d --sizem=1000 --sizen=1000 --sizennz=4000
--verify=1
```

Chapter 12 AOCL-LibMem

AOCL-LibMem is a Linux library of data movement and manipulation functions (such as memcpy() and strcpy()) highly optimized for AMD Zen micro-architecture. This library has multiple implementations of each function that can be chosen based on the application requirements as per alignments, instruction choice, threshold values, and tunable parameters. By default, this library will choose the best fit implementation based on the underlying micro-architectural support for CPU features and instructions.

This is the first release of the AOCL-LibMem library and it supports the following functions:

- memcpy
- mempcpy
- memmove
- memset
- memcmp

12.1 Building AOCL-LibMem for Linux

Minimum software requirements for compilation:

- GCC 11.1
- AOCC 3.2
- 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://developer.amd.com/amd-aocl/
2. Locate the *aocl-libmem* folder in the root directory.
3. Create build directory:

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

4. Configure for one of the following builds as required:

   – GCC

   ```
   Default
   $ cmake -D CMAKE_C_COMPILER=gcc ../aocl-libmem

   Enabling Tunable Parameters
   $ cmake -D CMAKE_C_COMPILER=gcc -D ENABLE_TUNABLES=Y ../aocl-libmem

   Compiling AVX512 on non-avx512 Machine
   $ cmake -D CMAKE_C_COMPILER=gcc -D ENABLE_TUNABLES=Y -D USE_AVX512=Y ../aocl-libmem
   ```

   – AOCC (Clang)

   ```
   Default
   $ cmake -D CMAKE_C_COMPILER=clang ../aocl-libmem

   Enabling Tunable Parameters
   $ cmake -D CMAKE_C_COMPILER=clang -D ENABLE_TUNABLES=Y ./aocl-libmem

   Compiling AVX512 on non-avx512 Machine
   $ cmake -D CMAKE_C_COMPILER=clang -D ENABLE_TUNABLES=Y -D USE_AVX512=Y ../aocl-libmem
   ```

5. Build:

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

6. Install:

   ```
   $ make install
   ```

After compilation, a shared library file *libaocl-libmem.so* will be generated and stored in *<build/lib> path*.

Note: *It is recommended not to load/run the AVX512 library on a non-AVX512 machine as it can lead to crash (invalid instructions).*

12.2 Running an Application

The applications must preload AOCL-LibMem to replace 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 16.

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 16. Application Implementations

Application Requirement	LIBMEM_OPERATION	Instructions	Side-effects
Vector unaligned source and de stination	[avx2|avx512],b,b	Load:VMOVDQU; Store:VMOVDQU	None
Vectoraligned source and desti nation	[avx2|avx512],y,y	Load:VMOVDQA; Store:VMOVDQA	Unaligned source and/or destination address will lead to crash
Vectoraligned source and unaligned destination	[avx2|avx512],y,[b|w|d|q|x]	Load:VMOVDQA; Store:VMOVDQU	None

Table 16. Application Implementations

Application Requirement	LIBMEM_OPERATION	Instructions	Side-effects
Vector unaligned source andali gned 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 17. 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 Linux library consisting of the basic cryptographic functions optimized for AMD “Zen” micro-architecture. This library has multiple implementations of different Advanced Encryption Standard (AES) cryptographic encryption/decryption ciphers and Secure Hash Algorithm 2 (SHA-2) hash functions.

This is the first release of the AOCL-Cryptography library with the following functions:

- AES encrypt/decrypt routines supporting the following cipher modes:
  - Cipher Block Chaining (CBC)
  - Cipher Feedback (CFB)
  - Output Feedback (OFB)
  - Counter (CTR)
  - Galois/Counter Mode (GCM)
  - Ciphertext Stealing Mode (XTS)

  All the mentioned cipher modes support 128, 192, and 256-bit key sizes.
- SHA-2 hash functions with the following digest sizes:
  - SHA-224
  - SHA-256
  - SHA-384
  - SHA-512

13.1 Requirements

- CMake 3.14
- GCC 11.1.0
- For more information on supported Linux operating systems, refer “Operating Systems” on page 13.

13.2 Using AOCL-Cryptography in a Sample Application

A few pointers for using AOCL-Cryptography in a sample application:

- For using the encrypt/decrypt routines, use the header file in the test application:

  include/alcp/alcp.h

  An example to use the cipher routines can be found in:

  aocl-crypto/examples/cipher

- For using the digest routines, use the header file:

  include/alcp/digest.h

  An example to use the digest routines can be found in:

  aocl-crypto/examples/digest

13.2.1 Compiling and Running AOCL-Cryptography Examples

Complete the following steps to compile and run the AOCL-Cryptography examples:

1. Download and untar the aocl-crypto package.
2. `cd amd-crypto`
3. `make`
4. To run example applications (for digest):

   ```
   LD_LIBRARY_PATH=<path where aocl libs are installed> ./bin/digest/sha2_384_example
   ```

13.2.2 Running OpenSSL Benchmarks Using AOCL-Cryptography Library

Use OpenSSL 3.0.0 to execute the following command:

```
openssl speed -provider <libname> -provider-path <path> -evp sha512
```

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 based compression and decompression methods.
- A unified standardized API set and the existing native APIs of the respective methods.
- Dynamic dispatcher feature that executes the most optimal function variant implemented using Function Multi-versioning and hence, offering a single optimized library portable across different x86 CPU architectures.

A test suite is provided for validation and performance benchmarking of the supported compression and decompression methods. The test suite also supports the benchmarking of IPP compression methods, such as lz4, lz4hc, and zlib on the Linux-based platforms.

14.1 Installation

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/*)

14.2 Running AOCL-Compression Test Bench on Linux

Test bench supports several options to validate, benchmark, or debug the supported compression methods. It uses the unified API set to invoke the compression methods supported by AOCL-Compression. It can also invoke and benchmark some of the IPP's compression methods.

To check the various options supported by the test bench, use one of the following commands:

```
aocl_compression_bench -h
Or
aocl_compression_bench --help
```

Use the following command for an example to run the test bench and validate the outputs from all the supported compression and decompression methods for a given input file:

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

Use the following command for an example to run the test bench and check the performance of a particular compression and decompression method for a given input file:

```
aocl_compression_bench -ezstd:5:0 -p <input filename>
```

Here, 5 is the level and 0 is the additional parameter to specify the custom window size for the ZSTD method.

To run the test bench with *error/debug/trace/info logs*, use the command:

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

Here, you can pass `-v` with a number such as `v<n>` that can take the following values:

- 1 for Error (default)
- 2 for Info
- 3 for Debug
- 4 for Trace

To test and benchmark the performance of IPP's compression methods, use the test bench option `-c` along with the other relevant options (as explained above).

Currently, IPP's lz4, lz4hc, 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 and zlib-1.2.11 source packages.
3. Apply IPP's lz4 and zlib patch files as follows:

   ```
   patch -p1 <"path to corresponding patch file">
   ```
4. Build the patched IPP lz4 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 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>
   ```

14.3 Running AOCL-Compression Test Bench on Windows

Test bench on Windows supports all the user options as on Linux, except for the `-c` option to link and test the IPP's compression methods. For more information, refer to "Running AOCL-Compression Test Bench on Linux" on page 106.

To set and launch the test bench with a specific user option:

1. Go to project aocl_compression_bench > **Properties** > **Debugging**.
2. Specify the user options and the input test file.

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;
```

```
//Interface data structure
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:0-NA,1-SSE2,2-AVX,3-AVX2,4-AVX512
    int printDebugLogs;  //print debug logs
    //size_t chunk_size; //Unused variable
} aocl_compression_desc;
```

14.4.3 Native APIs

```
//bzip2 Interface API to compress data
int BZ2_bzBuffToBuffCompress(
char*         dest,
unsigned int* destLen,
char*         source,
unsigned int  sourceLen,
int           blockSize100k,
int           verbosity,
int           workFactor
   );

//bzip2 Interface API to decompress data
int BZ2_bzBuffToBuffDecompress (
char*         dest,
unsigned int* destLen,
char*         source,
unsigned int  sourceLen,
int           small,
int           verbosity
   );
```

```
//lz4 Interface API to compress data
int LZ4_compress_default(
      const char* src,
char* dst,
int srcSize,
int dstCapacity
);

//lz4 Interface API to decompress data
int LZ4_decompress_safe  (
const char* src,
char* dst,
int compressedSize,
int dstCapacity
);

//lz4hc Interface API to compress data
int LZ4_compress_HC(
      const char* src,
char* dst,
int srcSize,
int dstCapacity,
int compressionLevel
);

//lz4hc Interface API to decompress data
int LZ4_decompress_safe (
const char* src,
char* dst,
int compressedSize,
int dstCapacity
);

//lzma Interface API to compress data
int LzmaEncode(
Byte *dest, SizeT *destLen, const Byte *src, SizeT srcLen,
const CLzmaEncProps *props, Byte *propsEncoded, SizeT *propsSize, int writeEndMark,
ICompressProgress *progress, ISzAllocPtr alloc, ISzAllocPtr allocBig
);

//lzma Interface API to decompress data
int LzmaDecode(
Byte *dest, SizeT *destLen, const Byte *src, SizeT *srcLen,
const Byte *propData, unsigned propSize, ELzmaFinishMode finishMode,
ELzmaStatus *status, ISzAllocPtr alloc
);
```

```
//snappy Interface API to compress data
void RawCompress(
const char* input,
size_t input_length,
char* compressed,
size_t* compressed_length
);

//snappy Interface API to decompress data
bool RawUncompress(
const char* compressed, size_t compressed_length,
char* uncompressed
);

//zlib Interface API to compress data
Int compress2(
unsigned char *dest,   unsigned long *destLen,
const unsigned char *source, unsigned long sourceLen,
int level
);

//zlib Interface API to decompress data
int uncompress(
unsigned char *dest,   unsigned long *destLen,
const unsigned char *source, unsigned long sourceLen
);

//zstd Interface API to compress data
size_t ZSTD_compress_advanced(
ZSTD_CCtx* cctx,
void* dst, size_t dstCapacity,
const void* src, size_t srcSize,
const void* dict,size_t dictSize,
ZSTD_parameters params
);

//zstd Interface API to decompress data
size_t ZSTD_decompressDCtx(
ZSTD_DCtx* dctx,
void* dst, size_t dstCapacity,
const void* src, size_t srcSize
);
```

14.4.4 Example Test Program

The following test program shows the sample usage and calling sequence of aocl-compression APIs to compress and decompress a test input:

```
#include <stdio.h>
#include <stdint.h>
#include <stddef.h>
#include <string.h>
#include "api/api.h"

int main (int argc, char **argv)
{
aocl_compression_desc aocl_compression_ds;
aocl_compression_desc *aocl_compression_handle = &aocl_compression_ds;
FILE *inFp = NULL;
int file_size = 0;
char *inPtr = NULL, *compPtr = NULL, *decompPtr = NULL;
int64_t resultComp = 0, resultDecomp = 0;

if (argc < 2)
{
    printf("Provide input test file path\n");
    return -1;
}
inFp = fopen(argv[1], "rb");
fseek(inFp, 0L, SEEK_END);
file_size = ftell(inFp);
rewind(inFp);

//One of the compression methods as per aocl_compression_type
aocl_compression_type method = LZ4;

aocl_compression_handle->level = 0;
aocl_compression_handle->optVar = 0;
aocl_compression_handle->printDebugLogs = 0;
aocl_compression_handle->inSize = file_size;
aocl_compression_handle->outSize = (file_size + (file_size / 6) + (16*1024));
inPtr = (char *)calloc(1, aocl_compression_handle->inSize);
compPtr = (char *)calloc(1, aocl_compression_handle->outSize);
decompPtr = (char *)calloc(1, aocl_compression_handle->inSize);
aocl_compression_handle->inBuf = inPtr;
aocl_compression_handle->outBuf = compPtr;
file_size = fread(inPtr, 1, file_size, inFp);

//1. setup and create a handle
aocl_llc_setup(aocl_compression_handle, method);

//2. compress
resultComp = aocl_llc_compress(aocl_compression_handle, method);
if (resultComp <= 0)
{
           printf("Compression: failed\n");
           goto error_exit;
}
```

```
 //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;
}

//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 *api.h* header file and link the *libaocl_compression.so* file as follows:

```
gcc test.c -I< api.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:

- SNAPPY_MATCH_SKIP_OPT: If this configure option is enabled, AOCL optimized Snappy method uses an improved heuristic-based match skipping approach to improve the compression speed with a minor trade off in the compression ratio.
- AOCL_LZ4_OPT_PREFETCH_BACKWARDS: If this configure option is enabled, prefetching is used in the backward direction for extending the match. This helps improve the performance of the compression process depending upon the test input data. In other cases, when the test input conditions do not satisfy the conditions assumed by this optimization, a minor performance drop may be observed.

Chapter 15 Linking- AOCL to Applications

This section provides examples of how AOCL can be linked with the HPL benchmark and MUMPS sparse solver library.

15.1 High-performance LINPACK Benchmark (HPL)

HPL is a software package that solves a (random) dense linear system in double precision (64-bits) arithmetic on distributed memory computers. It is a LINPACK benchmark that measures the floating-point rate of execution for solving a linear system of equations.

To build an HPL binary from the source code, edit the MPxxx and LAxxx directories in your architecture-specific Makefile to match the installed locations of your MPI and Linear Algebra library. For AOCL-BLIS, use the F77 interface with F2CDEFS = -DAdd__ -DF77_INTEGER=int -DStringSunStyle.

Use the multi-threaded AOCL-BLIS with the following configuration for an optimal performance:

```
./configure --enable-cblas -t openmp --disable-sup-handling --prefix=<path> auto
```

Setup HPL.dat before running the benchmark.

15.1.1 Configuring HPL.dat

HPL.dat file contains the configuration parameters. The important parameters are Problem Size, Process Grid, and BlockSize.

- Problem Size (N) — For best results, the problem size must be set large enough to use 80-90% of the available memory.
- Process Grid (P and Q) — P x Q must match the number of MPI ranks. P and Q must be as close to each other as possible. If the numbers cannot be equal, Q must be larger.
- BlockSize (NB) — HPL uses the block size for the data distribution and for the computational granularity. Set NB=240 for an optimal performance.
- Set BCASTs=2 — Increasing-2-ring (2rg) broadcast algorithm gives a better performance than the default broadcast algorithm.

15.1.2 Running the Benchmark

The combination of multi-threading (through OpenMP library) and MPI is important to configure for optimal performance. Set the number of MPI tasks to number of L3 caches in the system for optimal performance.

The HPL benchmark typically produces a better single node performance number with the following configurations depending on which generation of AMD EPYC^{TM} processor is used:

- 2nd Gen AMD EPYC™ Processors (codenamed “Rome”)

  A dual socket AMD EPYC 7742 system consists of 32 CCXs, each having an L3 cache and a total of 2 x 64 cores (four cores per CCX). For maximum performance, use 32 MPI ranks with 4 OpenMP threads. Each MPI rank is bonded to 1 CCX and 4 threads per L3 cache.

  Set the following flags while building and running the tests:

  ```
  export BLIS_IC_NT=4
  export BLIS_JC_NT=1
  ```

  Execute the following command to run the test:

  ```
  mpirun -np 32 --report-bindings --map-by ppr:1:l3cache,pe=4 -x OMP_NUM_THREADS=4 -x
  OMP_PROC_BIND=TRUE -x OMP_PLACES=cores ./xhpl
  ```

  BLIS_IC_NT and BLIS_JC_NT parameters are set for DGEMM parallelization at each shared L3 cache to improve the performance further.

- 3rd Gen AMD EPYC™ Processors (codenamed “Milan”)

  The number of MPI ranks and maximum thread count per MPI rank depends on the specific EPYC SKU. For better performance, bind each MPI rank to a CCX, if there are 4 OpenMP threads. However, if 8 threads are used, then you should specify CCD instead of CCX.

  Set the following flags while building and running the tests:

  ```
  export BLIS_IC_NT=8
  export BLIS_JC_NT=1
  ```

  Execute the following command to run the test:

  ```
  mpirun -np 16 --report-bindings --map-by ppr:1:l3cache,pe=8 -x OMP_NUM_THREADS=8 -x
  OMP_PROC_BIND=TRUE -x OMP_PLACES=cores ./xhpl
  ```

15.2 MUMPS Sparse Solver Library

MUltifrontal Massively Parallel Solver (MUMPS: *http://mumps-solver.org/*) is an open-source package for solving systems of linear equations of the form:

$Ax = b$

Where, A is a square sparse matrix that can be one of the following on distributed memory computers:

- Unsymmetric
- Symmetric positive definite
- General symmetric

MUMPS implements a direct method based on a multi-frontal approach which performs the Gaussian factorization:

$A = LU$

Where, L is a lower triangular matrix and U an upper triangular matrix.

If the matrix is symmetric then the factorization:

A = LDLT

Where, D is a block diagonal matrix performed.

The system Ax = b is solved in the following steps:

1. Analysis

   During an analysis, preprocessing including re-ordering and a symbolic factorization are performed. This depends on the external libs METIS, SCOTCH, and PORD (inside MUMPS source). Apre denotes the preprocessed matrix.

2. Factorization

   During the factorization, Apre = LU or Apre = LDLT, depending on the symmetry of the preprocessed matrix, is computed. The original matrix is first distributed (or redistributed) onto the processors depending on the mapping computed during the analysis. The numerical factorization is then a sequence of dense factorization on the frontal matrices.

3. Solution

   The solution xpre of:

   LUxpre = bpre or LDLT xpre = bpre

   Where, xpre and bpre are the transformed solution x and right-hand side b respectively. They are associated to the preprocessed matrix Apre and obtained through the forward elimination step:

   Ly = bpre or LDy = bpre

   Followed by the backward elimination step:

   Uxpre = y or L T xpre = y .

   The solution xpre is finally processed to obtain the solution x of the original system Ax = b.

The AOCL libraries can be integrated with the MUMPS sparse solver to perform highly optimized linear algebra operations on AMD “Zen”-based processors.

15.2.1 Enabling AOCL with MUMPS

15.2.1.1 Using Spack On Linux

Complete the following steps to enable AOCL with MUMPS on Linux:

1. Set up Spack on the target machine.

2. Link the AOCL libraries AOCL-BLIS, AOCL-libFLAME, and AOCL-ScaLAPACK while installing MUMPS. Use the following Spack commands to install MUMPS with:
   – gcc compiler:
     ```
     $ spack install mumps ^amdblis ^amdlibflame ^amdscalapack
     ```
   – aocc compiler:
     ```
     $ spack install mumps ^amdblis ^amdlibflame ^amdscalapack %aocc
     ```
   – To use an external reordering library (for example, METIS), run the following command:
     ```
     $ spack install mumps ^metis  ^amdblis ^amdlibflame ^amdscalapack
     ```

15.2.1.2 On Windows

GitHub URL: *https://github.com/amd/mumps-build*

Prerequisites

Ensure that the following prerequisites are met:

- CMake and Ninja Makefile Generator — Ensure that Ninja is installed/updated in the Microsoft Visual Studio installation folder:

  C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\Common7\IDE\CommonExtensions\Microsoft\CMake\Ninja

- Download the latest Binary Ninja from the URL:

  https://github.com/ninja-build/ninja/releases

- Intel® oneAPI toolkit must include C, C++, Fortran Compilers, and MPI. For more information, refer Intel documentation (*https://software.intel.com/content/www/us/en/develop/articles/oneapi-standalone-components.html#vtune*).
- Pre-built AOCL libraries for AOCL-BLIS, AOCL-libFLAME, and AOCL-ScaLAPACK.
- If reordering library is METIS, complete the following steps:
  a. Download the pre-built METIS library from SuiteSparse public repository (*https://github.com/grup-gu/SuiteSparse.git*).
  b. Build METIS library from the *metis* folder:
     ```
     cd SuiteSparse\metis-5.1.0
     ```
  c. Define **IDXTYPEWIDTH** and **REALTYPEWIDTH** to 32 or 64 based on the required integer size in *metis/include/metis.h*.
  d. Configure:
     ```
     cmake S . -B ninja_build_dir -G "Ninja" -DBUILD_SHARED_LIBS=OFF
     -DCMAKE_BUILD_TYPE=Release -DCMAKE_VERBOSE_MAKEFILE:BOOL=ON
     ```
  e. Build the project:
     ```
     cmake --build ninja_build_dir --verbose
     ```

  The library *metis.lib* is generated in *ninja_build_dir\lib*.

- Boost libraries on Windows:
  - Required to read the *.mtx* files efficiently and quickly
  - Essential for the test application *aocl_amd.cpp* that links to MUMPS libraries and measures the performance for an Symmetric Positive Definite (SPD) .mtx file
  - Download sources and bootstrap as instructed at the following URL:

    https://www.boost.org/doc/libs/1_55_0/more/getting_started/windows.html
  - Define **BOOST_ROOT** in *tests/CMakeLists.txt*

Building MUMPS Sources

Complete the following steps to build the MUMPS sources on Windows:

1. Checkout the MUMPS build repository from AOCL GitHub (*https://github.com/amd/mumps-build*).
2. Open Intel oneAPI command prompt for Intel 64 for Microsoft Visual Studio 2019 from Windows search box.
3. Edit the default options in *options.cmake* in *mumps/cmake/.*
4. Remove any build directory if it exists already.

5. Configure the MUMPS project using Ninja:

```
cmake S . -B ninja_build_dir -G "Ninja" -DENABLE_AOCL=ON -DENABLE_MKL=OFF -DBUILD_TESTING=ON
-DCMAKE_INSTALL_PREFIX="</mumps/install/path>" -Dscotch=ON -Dopenmp=ON -DBUILD_SHARED_LIBS=OFF
-Dparallel=ON -DCMAKE_VERBOSE_MAKEFILE:BOOL=ON -DCMAKE_BUILD_TYPE=Release
-DUSER_PROVIDED_BLIS_LIBRARY_PATH="<path/to/blis/library/path>"
-DUSER_PROVIDED_BLIS_INCLUDE_PATH="<path/to/blis/headers/path>"
-DUSER_PROVIDED_LAPACK_LIBRARY_PATH="<path/to/libflame/library/path>"
-DUSER_PROVIDED_LAPACK_INCLUDE_PATH="<path/to/libflame/headers/path>"
-DUSER_PROVIDED_SCALAPACK_LIBRARY_PATH="<path/to/scalapack/library/path>"
-DUSER_PROVIDED_METIS_LIBRARY_PATH="<path/to/metis/library/path>"
-DUSER_PROVIDED_METIS_INCLUDE_PATH="<path/to/metis/include/path>"
-DCMAKE_C_COMPILER=" icl.exe" -DCMAKE_CXX_COMPILER=" icl.exe"
-DCMAKE_Fortran_COMPILER="ifort.exe" -DBOOST_ROOT="<path/to/boost_1_77_0>" -Dintsize64=OFF -
DUSER_PROVIDED_IMPILIB_ILP64_PATH="<path/to/64-bit/Intel IMPI Library>"
-DMUMPS_UPSTREAM_VERSION="5.4.1"
```

The following options are enabled in the command:

– **-DENABLE_AOCL=ON**: <Enable AOCL Libraries>
– **-DENABLE_MKL=OFF**: <Enable MKL Libraries>
– **-DBUILD_TESTING=ON**: <Enable Mumps linking to test application to test>
– **-Dscotch=ON**: <Enable Metis Library for Reordering>
– **-Dopenmp=ON**: <Enable Multithreading using openmp>
– **-Dintsize64=OFF**: <Enable LP64 i.e., 32-bit integer size>
– **-DBUILD_SHARED_LIBS=OFF**: <Enable Static Library>
– **-Dparallel=ON**: <Enable Multithreading>
– **-DCMAKE_VERBOSE_MAKEFILE:BOOL=ON**: <Enable verbose build log>
– **-DCMAKE_BUILD_TYPE= Release**: <Enable Release build>
– **-DUSER_PROVIDED_BLIS_LIBRARY_PATH=** “<path/to/blis/lib>”
– **-DUSER_PROVIDED_BLIS_INCLUDE_PATH=** “<path/to/blis/header>”
– **-DUSER_PROVIDED_LAPACK_LIBRARY_PATH=** “<path/to/libflame/lib >”
– **-DUSER_PROVIDED_LAPACK_INCLUDE_PATH=** “<path/to/libflame/include/header
– **-DUSER_PROVIDED_SCALAPACK_LIBRARY_PATH=** “<path/to/scalapack/lib
– **-DUSER_PROVIDED_METIS_LIBRARY=** “<Metis/library/with/absolute/path >”
– **-DUSER_PROVIDED_METIS_LIBRARY_PATH=** “<path/to/metis/lib>”
– **-DUSER_PROVIDED_METIS_INCLUDE_PATH=** “<path/to/metis/header>”
– **-DCMAKE_C_COMPILER=** “<intel c compiler>”
– **-DCMAKE_Fortran_COMPILER=** “<intel fortran compiler>”
– **-DBOOST_ROOT=** “<path/to/BOOST/INSTALLATION>”
– **-DUSER_PROVIDED_IMPILIB_ILP64_PATH=**“<path/to/64-bit/Intel IMPI Library>”
– **-DMUMPS_UPSTREAM_VERSION**=“<valid/supported mumps source versions: 5.4.1 and 5.5.0>”

6. Toggle/Edit the options in step 5 to get:
   a. Debug or Release build
   b. LP64 or ILP64 libs
   c. AOCL or MKL Libs
7. Build the project:

```
cmake --build ninja_build_dir --verbose
```

8. Run the executable in *ninja_build_dir\tests:*

```
mpiexec -n 2 --map-by L3cache --bind-to core Csimple.exe
mpiexec -n 2 --map-by L3cache --bind-to core amd_mumps_aocl sample.mtx
```

Chapter 16 AOCL Tuning Guidelines

This section provides tuning recommendations for AOCL.

16.1 AOCL-BLIS Thread Control

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

16.1.1 AOCL-BLIS Initialization

During AOCL-BLIS 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) BLIS 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-BLIS 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-BLIS initialization:

Table 18. Sample Scenarios - 1

Sample Pseudo Code for Application	Sample Command Executed	Number of Threads Set During AOCL-BLIS Initialization	Remarks

`int main()`
`{`
`    ////pseudo code to use OpenMP API to set number of threads //////`
 	
`omp_set_num_threads(16);`
`    dgemm_( );`
`    ////////////`
`    return 0;`
`}`
	`$ 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.

`int main()`
`{`
`    ////pseudo code //////`
`    dgemm_( );`
`    ////////////`
`    return 0;`
`}`
	`$ 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-BLIS 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 19. Sample Scenarios - 2

Sample Pseudo Code for Application	Sample Command Executed	m Value in Sequence of Execution	Number of Threads for this BLAS Call	Remarks

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;
}
	$./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

The Runtime Thread Control feature details are as follows:

- The Runtime Thread Control feature facilitates the application to allocate different number of threads to BLIS from the number of threads application is using.
- Previously, when an application sets BLIS_NUM_THREADS with a valid value, BLIS internally called omp_set_num_threads() API with same value. Due to this, the application would not be able to differentiate between the number of threads used in BLIS library and the application.

- Currently, an application can choose either BLIS_NUM_THREADS environment variable or bli_thread_set_num_threads(nt) API for BLIS and OpenMP APIs/environment variables respectively.
- If BLIS_NUM_THREADS is set with a valid value, it will be used in the subsequent parallel regions unless bli_thread_set_num_threads() API is used by the application to modify the desired number of threads during BLIS API execution.
- Once BLIS_NUM_THREADS environment variable or bli_thread_set_num_threads(nt) API is used by the application, BLIS module would always give precedence to these values. BLIS API would not consider the values set using OpenMP API omp_set_num_threads(nt) API or OMP_NUM_THREADS environment variable.
- If BLIS_NUM_THREADS is not set,if the application is multithreaded and issued omp_set_num_threads(nt) with the desired number of threads, omp_get_max_threads() API will fetch the number of threads set earlier.
- If BLIS_NUM_THREADS is not set, omp_set_num_threads(nt) is not called by the application, but only OMP_NUM_THREADS is set, omp_get_max_threads() API will fetch the value of OMP_NUM_THREADS.
- If both environment variables are not set or if they are set with invalid values and omp_set_num_threads(nt) is not issued by application, omp_get_max_threads() API will return the number of the cores in the current context.
- BLIS will initialize rntm > num_threads with the value derived based on the above conditions. If omp_set_nested is false and the application calls BLIS APIs from parallel threads, BLIS APIs will run in a sequential manner. However, if nested parallelism is enabled, BLIS APIs can run on parallel threads internally.
- Order of precedence used for the number of threads:
  a. Value set using bli_thread_set_num_threads(nt) by the application.
  b. Valid value set for the environment variable BLIS_NUM_THREADS.
  c. omp_set_num_threads(nt) issued by the application.
  d. Valid value set for the environment variable OMP_NUM_THREADS.
  e. The number of cores.
- If nt is not a valid value for omp_set_num_threads(nt) API, the number of threads would be set to 1. omp_get_max_threads() API will return 1.
- OMP_NUM_THREADS environment variable is applicable only when OpenMP is enabled.
- Existing precedence of BLIS_*_NT environment variables and the decision of optimal number of threads obtained from the AOCL-BLIS Tuning Features, for example, AOCL Dynamic over the number of threads set by the application during BLIS initialization or runtime remains as it is.

16.2 AOCL Dynamic

The AOCL dynamic feature enables AOCL-BLIS 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 20. AOCL Dynamic

AOCL Dynamic	BLIS_NUM_THREADS	Number of Threads Used by AOCL-BLIS
Disabled	Unset	Number of Cores.
Disabled	Set	BLIS_NUM_THREADS
Enabled[a]	Unset	Number of threads determined by AOCL Dynamic.
Enabled[a]	Set	Minimum of BLIS_NUM_THREADS or the number of threads determined by AOCL.

a. The AOCL dynamic feature currently supports only DGEMM, DGEMMT, DTRSM, DTRMM, and DSYRK APIs. For the other APIs, the threads selection will be same as when AOCL Dynamic is disabled.

16.2.1 Limitations

The AOCL Dynamic feature has the following limitations:

- Support only for OpenMP Threads
- Supports only DGEMM, DGEMMT, DTRSM, and DSYRK APIs
- Specifying the number of threads more than the number of cores may result in deteriorated performance because of over-utilization of cores

16.3 AOCL-BLIS DGEMM Multi-thread Tuning

A AOCL-BLIS 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-BLIS 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-BLIS 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-BLIS 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-BLIS 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-BLIS.

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-BLIS source directory *\\blis\config\zen* defines certain macros that help control the block sizes used by AOCL-BLIS.

The required tuning settings vary depending on the number threads that the application linked to BLIS 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-BLIS with multi-thread option as mentioned in “Multi-thread AOCL-BLIS” on page 21.
3. Link the generated AOCL-BLIS 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-BLIS with the multi-thread option as mentioned in “Multi-thread AOCL-BLIS” on page 21.
3. Link the generated AOCL-BLIS 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-BLIS DGEMM Block-size Tuning

AOCL-BLIS DGEMM performance is largely impacted by the block sizes used by AOCL-BLIS. 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-BLIS 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-BLIS GitHub repository (*https://github.com/amd/blis*) is set to extract the best performance for such HPC applications/benchmarks, which use single-threaded AOCL-BLIS and run in multi-instance mode on AMD EPYC^{TM} 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^{TM} Processors (codenamed "Rome")

1. Open the file *bli_family_zen2.h* in the AOCL-BLIS source:

   ```
   $ cd “config/zen2/ bli_family_zen2.h”
   ```

2. For applications/benchmarks running in multi-instance mode and using multi-threaded AOCL-BLIS, 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-BLIS:

   ```
   #define AOCL_BLIS_MULTIINSTANCE        0
   ```

1st Gen AMD EPYC^{TM} Processors (codenamed "Naples")

1. Open the file *bli_cntx_init_zen.c* under the AOCL-BLIS 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^{TM} are defined in the file *config/zen/bli_cntx_init_zen.c:*

   ```
   bli_blksz_init_easy( &blkszs[ BLIS_MC ],   144,  240,   144,    72 );
   bli_blksz_init_easy( &blkszs[ BLIS_KC ],   256,  512,   256,   256 );
   bli_blksz_init_easy( &blkszs[ 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^{TM} are defined in the file *config/zen/bli_cntx_init_zen.c*:

   ```
   bli_blksz_init_easy( &blkszs[ BLIS_MC ],   144,  510,   144,    72 );
   bli_blksz_init_easy( &blkszs[ BLIS_KC ],   256,  1024,   256,   256 );
   bli_blksz_init_easy( &blkszs[ BLIS_NC ],  4080,  4080,  4080,  4080 );
   ```

16.5 Performance Suggestions for Skinny Matrices

AOCL-BLIS 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-libFLAME Multi-threading

From AOCL 4.0 release, AOCL-libFLAME supports multi-threading using OpenMP in selected APIs. This feature is enabled by default when AOCL-libFLAME is compiled with `--enable-amd-flags` or `--enable-amd-aocc-flags`. However, you can disable multi-threading by setting `--enable-multithreading=no`.

The selected LAPACK interface APIs that support multi-threading automatically choose optimal number of threads. However, you can explicitly set the number of threads through the environment variable or OpenMP runtime APIs. In such a scenario, the number of threads is selected as follows:

Thread Criteria	Threads Used by API
User specified threads > AOCL-libFLAME computed optimal threads	AOCL-libFLAME computed optimal threads
User specified threads < AOCL-libFLAME 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^{TM} CPUs.

  This is the mandatory master optimization switch that must be set for enabling any other optional configure options, such as:

  - `--enable-amd-mpifft`
  - `--enable-amd-mpi-vader-limit`
  - `--enable-amd-trans`
  - `--enable-amd-fast-planner`
  - `--enable-amd-top-n-planner`
  - `--enable-amd-app-opt`
  - `--enable-dynamic-dispatcher`
- When enabling the AMD CPU specific improvements with the configure option `--enable-amd-opt`, do not use the configure option `--enable-generic-simd128` or `--enable-generic-simd256`.
- An optional configure option `--enable-amd-trans` is provided and it may benefit the performance of transpose operations in the case of very large FFT problem sizes. This feature is to be used only when running in single-thread and single instance mode.
- Use the configure option `--enable-amd-mpifft` to enable MPI FFT related optimizations. This is provided as an optional parameter and will benefit most of the MPI problem types and sizes.
- An optional configure option `--enable-amd-mpi-vader-limit` that controls enabling of AMD's new MPI transpose algorithms is supported. When using this configure option, you must set `--mca btl_vader_eager_limit` appropriately (current preference is 65536) in the MPIRUN command.

- You can enable AMD optimized fast planner using the optional configure option `--enable-amd-fast-planner`. You can use this option to reduce the planning time without much trade-off in the performance. It is supported for single and double precisions.
- To minimize single-threaded run-to-run variations, you can enable the planner feature Top N planner using configure option `--enable-amd-top-n-planner`. It works by employing WISDOM feature to generate and reuse a set of top N plans for the given size (wherein the value of N is currently set to 3). It is supported for only single-threaded execution runs.
- For best performance, use the `PATIENT` planner flag of FFTW.

  A sample running of FFTW bench test application with `PATIENT` planner flag is as follows:

  ```
  $ ./bench -opatient -s icf65536
  ```

  Where, `-s` option is for speed/performance run and icf options stand for in-place, complex data-type, and forward transform.
- When configured with `--enable-openmp` and running multi-threaded test, set the OpenMP variables as:

  ```
  set OMP_PROC_BIND=TRUE
  OMP_PLACES=cores
  ```

  Then, run the test bench executable binary using numactl as follows:

  ```
  numactl --interleave=0,1,2,3 ./bench -opatient -onthreads=64 -s icf65536
  ```

  Where, numactl --interleave=0,1,2,3 sets the memory interleave policy on nodes 0, 1, 2, and 3.
- When running MPI FFTW test, set the appropriate MPI mapping, binding, and rank options.

  For example, to run 64 MPI rank FFTW on a 64-core AMD EPYC^{TM} 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 135.
- 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. It is supported for GCC compiler and Linux based systems for now. 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://developer.amd.com/amd-aocl/*).

Chapter 18 References

The following URLs have been used as references for this document:

- *https://developer.amd.com/amd-aocl/*
- *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^{TM} Processors (codenamed “Naples”), CPU Core AMD “Zen”
      – CPU Family: 23
      – Model: Values in the range <1 – 47>
   b. For 2nd Gen AMD EPYC^{TM} Processors (codenamed “Rome”), CPU Core AMD “Zen2”
      – CPU Family: 23
      – Model: Values in the range <48 – 63>
   c. For 3rd Gen AMD EPYC^{TM} Processors (codenamed “Milan”), CPU Core AMD “Zen3”
      – CPU Family: 25
      – Model: Values in the range <1 – 15>
   d. For 4th Gen AMD EPYC^{TM} 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^{TM} Processors (codenamed “Naples”), CPU Core AMD “Zen”
      – CPU Family: 23
      – Model: Values in the range <1 – 47>
   b. For 2nd Gen AMD EPYC^{TM} Processors (codenamed “Rome”), CPU Core AMD “Zen2”
      – CPU Family: 23
      – Model: Values in the range <48 – 63>

c. For 3rd Gen AMD EPYC^{TM} Processors (codenamed "Milan"), CPU Core AMD "Zen3"
   - CPU Family: 25
   - Model: Values in the range <1 – 15>

d. For 4th Gen AMD EPYC^{TM} 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-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.