AMD Optimizing CPU Libraries User Guide

Version 3.0

Table of Contents

1. Introduction 5
2. Supported Operating Systems and Compliers 6
3. AOCL Installation 7
3.1. Build from Source 7
3.1.1. Master Installer and tar Packages of AOCL Binaries 7
3.1.2. Individual Library Packages 7
3.1.3. Debian and RPM Packages 8
4. BLIS library for AMD 9
4.1. Installation 9
4.1.1. Build BLIS from source 9
4.1.1.1. Single-thread BLIS 9
4.1.1.2. Multi-threaded BLIS 10
4.1.2. Using Pre-built binaries 11
4.2. Usage 11
4.2.1. BLIS - Running in-built test suite 12
4.2.2. Testing/Benchmarking of GEMM with Custom Input 12
4.2.3. BLIS Usage in FORTRAN 14
4.2.4. BLIS Usage in C through BLAS and CBLAS APIs 15
4.2.5. Using BLIS Shared Library 18
4.2.6. BLIS APIs 18
4.3. Function Call Tracing and Debug Logs in BLIS 19
4.3.1. Function Call Tracing 20
4.3.2. Debug Logging 21
4.3.3. Usages and Limitations 23
5. libFLAME Library for AMD 24
5.1. Installation 24
5.1.1. Build libFLAME from Source 24
5.1.2. Using Pre-built Binaries 25
5.2. Usage 25
6. FFTW Library for AMD 27
6.1. Installation 27
6.1.1. Build FFTW from Source 27

3. To build the library, type 28
4. To install the library in preferred installation path, type 28
5. To verify the installed library, type 28
6.1.2. Using Pre-built Binaries 28
6.2. Usage 29
7. AMD LibM 30
7.1. Installation 32
7.2. Compiling AMD LibM 32
7.3. Usage 33
8. AMD Optimized memcpy 35
8.1. Building AMD Optimized memcpy 35
8.2. Building an Application 35
8.3. Running the Application 35
9. ScaLAPACK library for AMD 36
9.1. Installation 36
9.1.1.1. Build ScaLAPACK from source 36
9.1.2. Using Pre-built Binaries 38
9.2. Usage 39
10. AMD Random Number Generator 40
10.1. Installation 40
10.2. Usage 40
11. AMD Secure RNG 41
11.1. Installation 41
11.2. Usage 41
12. AOCL-Sparse 43
12.1. Installation 43
12.1.1. Build aocl-sparse From Source 43
12.1.2. Simple Test 45
12.1.3. Using Pre-built Libraries 45
12.2. Usage 45
13. AOCL Spack Recipes 48
13.1. AOCL Spack Environment Setup 48
13.2. Install AOCL packages 48

13.2.1. Install amdblis Spack Package 48
13.2.2. Install amdlibflame Spack Package 48
13.2.3. Install amdfftw Spack Package 48
13.2.4. Install amdscalapack Spack Package 49
13.2.5. Install legacy AOCL versions 49
13.3. Useful Spack Commands 49
13.4. Uninstall AOCL Packages 49
14. Applications integrated to AOCL 50
14.1. High-performance LINPACK Benchmark (HPL) 50
15. AOCL Tuning Guidelines 52
15.1. BLIS DGEMM Multi-thread Tuning 52
15.1.1. Library Usage Scenarios 52
15.1.2. The application is multi-thread and the library are single-threads 53
15.1.3. Both the Application and the Library are multi-threads 53
15.1.4. Architecture Specific Tuning 53
15.2. BLIS DGEMM Block-size Tuning for Single and Multi-instance mode 54
15.3. Performance Suggestions for Skinny Matrices 56
15.4. AMD Optimized FFTW Tuning Guidelines 56
16. Appendix 57
16.1. Check AMD Server Processor Architecture 57
16.2. Application Notes 58
16.2.1. AMD Optimized FFTW 58
17. Technical Support and Forums 59
18. References 60

1. Introduction

AMD Optimizing CPU Libraries (AOCL) are a set of numerical libraries optimized for AMD EPYC™ processor family. This document provides instructions on installing and using all the AMD optimized libraries.

AOCL comprise of the following eight packages:

1. **BLIS (BLAS Library)** – BLIS is a portable open-source software framework for performing high-performance Basic Linear Algebra Subprograms (BLAS) functionality.
2. **libFLAME (LAPACK)** - libFLAME is a portable library for dense matrix computations, which provides the functionality present in the Linear Algebra Package (LAPACK).
3. **FFTW** – FFTW (Fast Fourier Transform in the West) is a comprehensive collection of fast C routines for computing the Discrete Fourier Transform (DFT) and various special cases .
4. **LibM (AMD Core Math Library)** - AMD LibM is a software library containing a collection of basic math functions optimized for x86-64 processor-based machines.
5. **ScaLAPACK** - 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.
6. **AMD Random Number Generator Library** - It is a pseudorandom number generator library.
7. **AMD Secure RNG** - The AMD Secure RNG is a library that provides APIs to access the cryptographically secure random numbers generated by the AMD hardware random number generator.
8. **AOCL-Sparse** - A library that contains basic linear algebra subroutines for sparse matrices and vectors optimized for AMD EPYC family of processors.

Additionally, AMD provides the following:

- Spack based recipes for installing BLIS, libFLAME, ScaLAPACK, and FFTW libraries
- AMD optimized memcpy library

For more information on the AOCL release and installers refer the AMD developer site.

For any issues or queries on the libraries, send an email to toolchainsupport@amd.com.

AOCL 3.0 includes several performance improvements for AMD Milan based microprocessor architecture in addition to Rome and Naples architecture. Refer Appendix Check AMD Server Processor Architecture to determine the underlying architecture of your AMD system.

2. Supported Operating Systems and Compliers

This release of AOCL has been validated on the following operating systems and compilers:

Operating Systems

- Ubuntu 20.04 LTS
- CentOS 8
- RHEL 8.3.1
- SLES 15 SP3

Compilers

- GCC 8.3.1, 9.2.1, 9.2, and 9.3
- AOCC 3.0

Glibc

- 2.28 and 2.31

MPI

- OpenMPI 3.1.4

3. AOCL Installation

AOCL can be installed using one of the following methods:

3.1. Build from Source

You can download the open source libraries of AOCL suite including BLIS, libFLAME, FFTW, ScaLAPACK, and aocl-sparse from GitHub and built from source:

- BLIS Source
- libFLAME Source
- FFTW Source
- ScaLAPACK Source
- aocl-sparse

Details on installing from source for each library is explained in the later sections.

3.1.1. Master Installer and tar Packages of AOCL Binaries

AOCL master installer is available in the '*Download*' section and it can be used to install the entire AOCL library suite.

3.1.1.1. Steps to Install Master Package

1. Download the AOCL TAR packages of AOCL Binaries to the target machine.
2. Use the command `tar -xvf <AOCL_PACKAGE.tar.gz>` to untar the package.
3. Locate the installer file "`install.sh`" in the package.
4. Run ./install.sh to install the AOCL package to the path "/home/<username>/aocl/3.0".
5. To install the AOCL package in a custom location, use the installer with option -t.
   For example, `./install.sh -t /home/amd/`

Note: Master installer can also be used to install individual library out of master package.
To install a particular library, use the option "-l" followed by Library name.
For example, `./install.sh -l blis`
Library names used:- blis , libflame, libm, scalapack, rng, secrng, sparse)

You can also install the individual library in a path of your choice.
For example, `./install.sh -t /home/amd -l blis`

3.1.2. Individual Library Packages

You can download the individual library binaries from the respective libraries page.
For example, BLIS and libFLAME tar packages are available in the following URL:
https://developer.amd.com/amd-aocl/blas-library/

3.1.3. Debian and RPM Packages

The Debian and RPM packages of AOCL are available in the '*Download*' section of the following URL: https://developer.amd.com/amd-aocl/
The package name used in following installation steps is based on 'gcc' build. It is applicable for the AOCC build when you replace 'gcc' with 'aocc'.

Installing AOCL Debian Package

Complete the following steps to install the AOCL Debian Package:

1. Download the AOCL 3.0 RPM package to the target machine.
2. Check the install path before installing.
   $ dpkg -c aocl-linux-gcc-3.0.0_1_amd64.deb
3. Install the package.
   Note: User requires sudo privilege
   ```
   $ sudo dpkg -i aocl-linux-gcc-3.0.0_1_amd64.deb
   ```
   Or
   ```
   $ sudo apt install ./aocl-linux-gcc-3.0.0_1_amd64.deb
   ```
4. Display the installed package information along with the package version and a short description.
   ```
   $ dpkg -s aocl-linux-gcc-3.0.0
   ```
5. List the contents of the package.
   ```
   $dpkg -L aocl-linux-gcc-3.0.0
   ```

Uninstalling the Debian package

```
$ sudo dpkg -r aocl-linux-gcc-3.0.0
```
(or)
```
$ sudo apt remove aocl-linux-gcc-3.0.0
```

Installing the AOCL RPM Package

1. Download the AOCL 2.2 RPM package to the target machine.
2. Install the package.
   ```
   $ sudo rpm -ivh aocl-linux-gcc-3.0.02-1.x86_64.rpm
   ```
   Note: You must have the sudo privileges to perform this step.
3. Display the installed package information along with the package version and a short description.
   ```
   $ rpm -qi aocl-linux-gcc-3.0.0-1.x86_64
   ```
4. List the contents of the package.
   ```
   rpm -ql aocl-linux-gcc-3.0.0-1
   ```

Uninstalling the AOCL RPM Package
```
$ rpm -e aocl-linux-gcc-3.0.0-1
```

4. BLIS library for AMD

BLIS is a portable open-source software framework for instantiating high-performance Basic Linear Algebra Subprograms (BLAS) such as, dense linear algebra libraries. The framework was designed to isolate the essential kernels of computation. When optimized, the kernels enable the optimized implementations of most commonly used and computationally intensive operations. Select kernels have been optimized for the AMD EPYC™ processor family by AMD and others.

AMD has the optimized version of BLIS supports C, FORTRAN, and C++ template interfaces for the BLAS functionalities.

4.1. Installation

You can install BLIS from source or pre-built binaries.

4.1.1. Build BLIS from source

Github URL: https://github.com/amd/blis

BLIS uses configure/make method to build it from the sources. It can be configured to be built in the following ways:

- **"auto":** This configuration generates a binary compatible with build machine architecture and is optimized for the CPU architecture of the build machine. This is useful when the user builds the library on the target system.
- **"zen":** This configuration generates binary compatible and optimized for AMD Zen (Naples) architecture. The architecture of the build machine is not relevant.
- **"zen2":** This configuration generates binary compatible and optimized for AMD Zen2 (Rome) architecture. The architecture of the build machine is not relevant.
- **"zen3":** This configuration generates binary compatible and optimized for AMD Zen3 (Milan) architecture. The architecture of the build machine is not relevant.
- **"amd64":** The library build using this configuration generates binary compatible and optimized to Zen, Zen2, and Zen3 architectures. The architecture of the build machine is not relevant. The architecture of the target machine is checked during the runtime, based on which, relevant optimizations are auto picked.

4.1.1.1. Single-thread BLIS

Following are the build instructions for a single threaded AMD BLIS:

1. git clone https://github.com/amd/blis.git

2. Depending on the target system and the build environment, you must enable/disable the appropriate configure options. The following steps provide instructions for compiling on AMD CPU core-based platforms. For a complete list of the options and their descriptions, use the command *./configure --help*.

3. Starting from the BLIS 2.1 release, the "auto" configuration option enables selecting the appropriate build configuration based on the target CPU architecture. For example, on Naples, "zen" config will be chosen and on Rome, "zen2" config will be chosen. BLIS 3.0 release extends it to choose "zen3" config for Milan.

   With GCC (default):
   ```
   $ ./configure --enable-cblas --prefix=<your-install-dir> auto
   ```

   With AOCC:
   ```
   $ ./configure --enable-cblas --prefix=<your-install-dir> CFLAGS="-DAOCL_F2C" CXXFLAGS="-DAOCL_F2C" CC=clang CXX=clang++ auto
   ```

4. Build the library using command "$ make"

5. To install the library on build machine use command "$ make install"

The BLIS binary (libblis) included in the AOCL master installer package is built with "zen3" config.

4.1.1.2. Multi-threaded BLIS

Here are the build instructions for multi-threaded AMD BLIS.

1. git clone https://github.com/amd/blis.git
2. Depending on the target system and build environment, you must enable/disable the appropriate configure options. The following steps provide instructions for compiling on AMD CPU core-based platforms. For a complete list of options and their descriptions, use the command ./configure –help.
3. Starting from BLIS 2.1 release, the "auto" configuration option, enables selecting the appropriate build configuration based on the target CPU architecture. For example, on Naples, "zen" config will be chosen and on Rome, "zen2" config will be chosen. BLIS 3.0 release extends it to choose "zen3" config for Milan.

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

   With AOCC
   ```
   $ ./configure --enable-cblas --enable-threading=[Mode] --prefix=<your-install-dir> CFLAGS="-DAOCL_F2C" CXXFLAGS="-DAOCL_F2C" CC=clang CXX=clang++ auto
   ```

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

4. Build the library using command "$ make"

5. To install the library on build machine use command "$ make install"

The multi-thread BLIS binary (libblis-mt) included in the AOCL master installer package is built with OpenMP threading mode and "`zen3`" config. For more information on multi-threaded implementation in BLIS refer GitHub BLIS Multi-threading Documentation..

4.1.2. Using Pre-built binaries

AMD optimized BLIS library binaries for Linux are available in the following URLs:
https://github.com/amd/blis/releases
https://developer.amd.com/amd-aocl/blas-library/

Also, the BLIS binary can be installed from the AOCL master installer tar file available in the following URL:
https://developer.amd.com/amd-aocl/

The master installer includes both the single threaded and multi-threaded BLIS binaries. Both BLIS binaries are built with "`zen2`" config. The multi-thread BLIS binary (libblis-mt) included in AOCL master installer package is built with OpenMP threading mode.

The tar file includes pre-built binaries of other AMD Libraries libFLAME, LibM, FFTW, aocl-sparse, ScaLAPACK, Random Number Generator, and AMD Secure RNG.

4.2. Usage

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

To execute the tests, navigate to the BLIS source directory using the following command:

```
$ make check
```

Execute BLIS C++ Template API tests as follows:

```
$ make checkcpp
```

Use by Applications

To use BLIS in your application, you must link the library while building the application.

For example:
With Static Library:

```
gcc test_blis.c -I<path-to-BLIS-header>  <path-toBLIS-library>/libblis.a -o test_blis.x
```

With Dynamic Library:

```
gcc test_blis.c -I<path-to-BLIS-header>  -L<path-toBLIS-library>/libblis.so -o test_blis.x
```

BLIS also includes a BLAS compatibility layer, which gives the application developers access to the BLIS implementations through a traditional FORTRAN BLAS API calls, that can be used in FORTRAN and C code. BLIS also provides a CBLAS API, which is a C-style interface for BLAS that can be called from C code.

4.2.1. BLIS - Running in-built test suite

The BLIS source directory contains a test suite to verify the functionality of 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 a detailed information, refer https://github.com/flame/blis/blob/master/docs/Testsuite.md

Running Test Suite

Execution the following command to invoke the test suite:

```
$ make test
```

The sample output of the execution is as follows:

```
$:~/blis$ 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!
```

4.2.2. Testing/Benchmarking of GEMM with Custom Input

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

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

Follow the steps below 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

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

Creating Input File

The input file accepts matrix sizes and strides in following format. Each dimension is separated by a space and each entry is separated by new line. For example: M K N CS_A CS_B CS_C

Note: This test application (test_gemm.c) assumes column-major storage of matrices.
Valid values of CS_A, CS_B and CS_C for GEMM operation *C = beta*C + alpha* A * B*, are
CS_A >= M
CS_B >= K
CS_C >= M

Running the Tests

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

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  GEMM_Algo
data_gemm_blis   200      100     100     200     200     200     27.211          S
data_gemm_blis    10        4       1     100     100     100      0.027          S
data_gemm_blis  4000     4000     400    4000    4000    4000     45.279          N
$ cat outputs.txt
m       k       n       cs_a    cs_b    cs_c    gflops  GEMM_Algo
   200    100     100     200     200     200    27.211          S
    10      4       1     100     100     100     0.027          S
  4000   4000     400    4000    4000    4000    45.279          N
```

4.2.3. BLIS Usage in FORTRAN

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 it and link with the BLIS library is also shown below the 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
```

Sample compilation command with gfortran compiler for the code above :

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

4.2.4. BLIS Usage in C through BLAS and CBLAS APIs

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

Using BLIS with BLAS API in C code

Shown below is the C version of the code listed above in FORTRAN. It uses the standard BLAS API.

The following process takes place:

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 code below:

```
// 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;
}
```

Sample compilation command with a gcc compiler for the code above: :

```
gcc BLAS_DGEMM_usage.c -Ipath/to/include/blis/ path/to/libblis.a
```

Using BLIS with CBLAS API

This section contains example application written in C code using CBLAS APIs for the DGEMM functionality.

The following process takes place:

1. CBLAS Layout option allows you choose between row-major and column-major layouts (row-major layout is used in the example, which is in line with C-style.)

2. The function arguments can be passed by the value too.

3. "cblas.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:

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

// 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;
```

```
}
```

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

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

4.2.5. Using BLIS Shared Library

1. BLIS can be built as a shared library. Complete the following steps to build a shared lib version of BLIS and link it with the user application. 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 -l blis
   -lm -o CBLAS_DGEMM_usage.x
   ```

3. Ensure that the shared library is available in the library load path. The, run the application using 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
   ```

4.2.6. BLIS APIs

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

API	Usages	Comments/Caveats
bli_info_get_version_str	Gets the version info of the running BLIS library	Returns the version string in the form "AOCL-3.0"
bli_info_get_enable_openmp bli_info_get_enable_pthreads bli_info_get_enable_threading	Returns if openmp/pthread or either of them is enabled or disabled.	Return true/false

`bli_thread_get_num_threads*`	Returns the global number of threads	Returns number of threads per operation
`bli_thread_set_num_threads( dim_t n_threads );*`	Sets the global number of threads	Sets the number of threads per operation
`bli_thread_set_ways( dim_t jc, dim_t pc, dim_t ic, dim_t jr, dim_t ir );*`	Sets the number of threads for different levels of parallelization	Using this API, you can specify the number of threads used for the different loops

Overriding Default Cache Block Sizes

The cache block sizes affect the DGEMM performance and can be overridden if the default sizes do not provide the “optimal performance” for experimental purposes. This section explains the APIs and steps for defining cache block sizes.

```
// Specify the block sizes for various data type
// This API takes type of panel and block sizes for float, double, complex and
double complex
// respectively.
// Set the size to 0 if you want to use the default value for that data type.

//                   Panel Type     ,   s,    d,    c,    z

bli_blksz_init_easy(&blkszs[BLIS_MC],   0,    72,   0,    72 );
bli_blksz_init_easy(&blkszs[BLIS_KC],   0,   256,   0,   256 );
bli_blksz_init_easy(&blkszs[BLIS_NC],   0,  4080,   0,  4080 );

bli_cntx_set_blkszs(
    BLIS_NAT, 3,
    BLIS_NC, &blkszs[BLIS_NC], BLIS_NR,
    BLIS_KC, &blkszs[BLIS_KC], BLIS_KR,
    BLIS_MC, &blkszs[BLIS_MC], BLIS_MR,
    cntx);
```

* Refer to this link for details.
https://github.com/flame/blis/blob/master/docs/Multithreading.md#specifying-multithreading

4.3. Function Call Tracing and Debug Logs in BLIS

AMD BLIS library provides inbuilt Debug and Trace feature:

Trace Log: This feature is used to identify the code path taken in terms of function call chain. It prints the information on the functions invoked and their order.

Debug Log: They are used to print the other debugging information, such as values of input parameters, content, and data structures.

Key Features

- Can be enabled/disabled at the compile time.
- When these features are disabled at compile time, they do not need 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.

4.3.1. Function Call Tracing

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

The function call tracing also needs recompilation and some source code modification to get it working. The following section explains the steps to enable and view the traces. The function call traces are printed in the human readable text format.

1. Enable the trace support.

    - Modify the source code to enable tracing.

      ```
      Open file <blis folder>/aocl_dtl/aocldtlcf.h
      ```

    - Change the following macro from 0 to 1:

      ```
      #define AOCL_DTL_TRACE_ENABLE       0
      ```

2. Configure the trace depth level.
    - Modify the source code to specify trace depth level.

      ```
      Open file <blis folder>/aocl_dtl/aocldtlcf.h
      ```

    - Change the following macro as required. To begin with Level 5, should be 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.
    - Library is built normally as explained in section 4.1.1 Build BLIS from source
    -
4. Generate the trace data.
    - Run the application, trace output files for each thread is generated in the current folder.
    - Figure 1: Sample run of function call tracing, shows a samplerunning call tracing function using the test_gemm application.

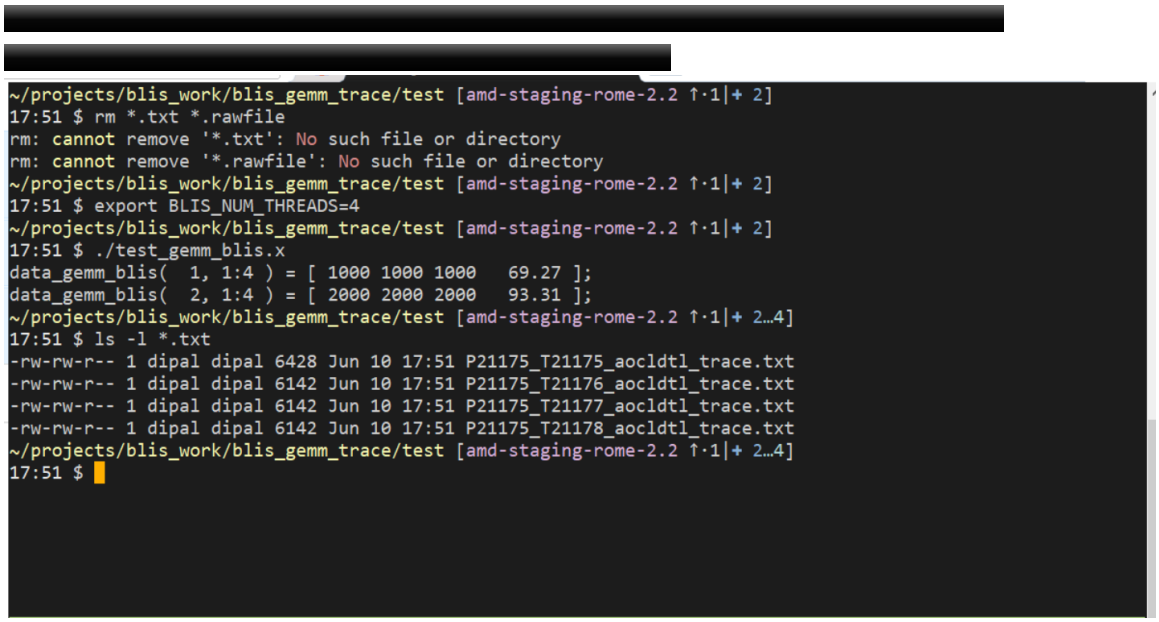

Figure 1: Sample run of function call tracing

As shown, the trace data for each thread is saved in the file with following any naming conventions. The .txt extension is used to signify the human 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, just open the file in any of the text editors.
   - The first column shows the level in call stack for the given function.
   - The trace is also independent according to the position of the function in the call stack.

4.3.2. Debug Logging

The debug logging works very similar to the function call tracing and uses the same infrastructure. However, it can be enabled independent of the trace feature to avoid the cluttering of the overall debugging information. This feature is primarily used to print the input values of BLIS APIs. However, it can be used to print any arbitrary debugging data.

1. Enable the debug logs.

   - Modify the source code to enable debug logging.

     `Open file <blis folder>/aocl_dtl/aocldtlcf.h`

   - Change the following macro from 0 to 1:

```
#define AOCL_DTL_LOG_ENABLE       0
```

2. Configure the debug logs depth level.
    - Modify the source code to specify the debug log depth level.

      ```
      Open file <blis folder>/aocl_dtl/aocldtlcf.h
      ```

    - Change the following macro as required., To begin with Level 5, should be 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 trace generation.

      ```
      #define AOCL_DTL_TRACE_LEVEL  AOCL_DTL_LEVEL_TRACE_5
      ```

3. Build the library, Library is built normally as explained in the section 4.1.1 Build BLIS from source
4. Generate the debug logs.
    - Run the application, the debug logs output files for each thread will be generated in the current folder.
    - Figure 2: A sample run with debug logs enabled shows the sample running of the BLIS with the debug logs enabled using the test_gemm application.
    - As shown, the debug logs for each thread are saved in the file with following naming conventions. The .txt extension is used to signify the human readable file.

      ```
      P<process id>_T<thread id>_aocldtl_log.txt
      ```

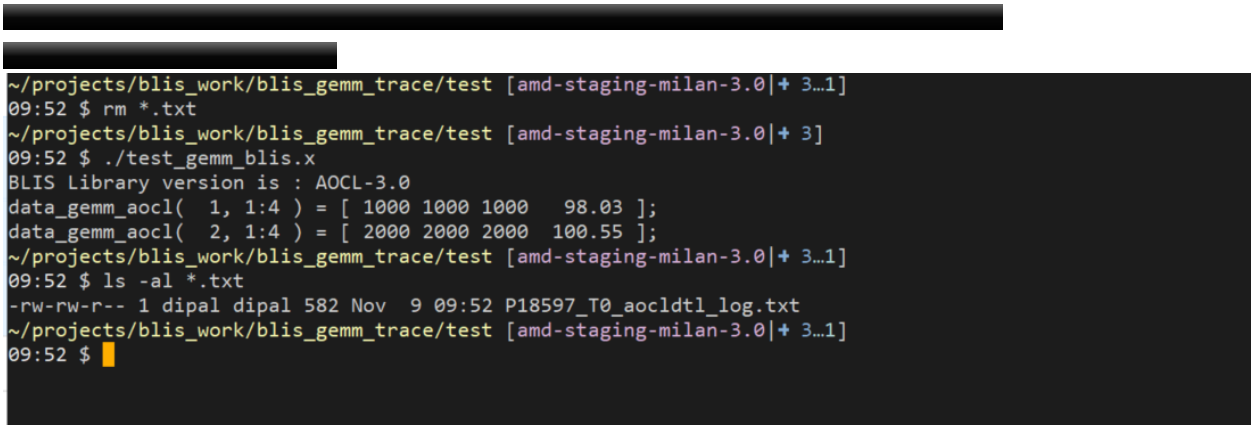

Figure 2: A sample run with debug logs enabled

5. View the trace data.
    - The output of the debug logs is in a readable format, just open the file in any of the text editors.
    - Figure 3: Sample output of debug logs showing input values of GEMM, shows the sample output for one of the threads of *test_gemm* application.

Figure 3: Sample output of debug logs showing input values of GEMM

4.3.3. Usages and Limitations

1. When tracing is enabled, performance will see a significant drop.
2. Only 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.
3. The call trace and debug logging is a resource dependent process, it can generate a large size of data. Based on the hardware configuration, that is, the diskspace, number of cores and threads needed for execution, it may result in a sluggish or non-responsive system.

5. libFLAME Library for AMD

libFLAME is a portable library for dense matrix computations, providing the functionality present in Linear Algebra Package (LAPACK). It includes a compatibility layer, FLAPACK, which includes the complete LAPACK implementation. The library provides scientific and numerical computing communities with a modern high-performance dense linear algebra library. It is extensible, easy to use, and available under an open-source license. libFLAME is a C-only implementation and does not depend on any external FORTRAN libraries including LAPACK. There is an optional backward compatibility layer, lapack2flame, that maps LAPACK routine invocations to their corresponding native C implementations in libFLAME. This allows the legacy applications to start taking advantage of libFLAME with virtually no changes to their source code.

Starting from AOCL 2.2 release, AMD optimized version of libFLAME is compatible with LAPACK 3.9.0 specification. In combination with the BLIS library, which includes optimizations for the AMD EPYC™ processor family, libFLAME enables running high performing LAPACK functionalities on AMD platform. AMD version of libFLAME supports C, FORTRAN, and C++ template interfaces for the LAPACK functionalities.

5.1. Installation

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

5.1.1. Build libFLAME from Source

Github link: https://github.com/amd/libflame

Note: Building libFLAME library does not require linking to BLIS or any other BLAS library. The applications which use libFLAME will have to link with BLIS (or other BLAS libraries) for the BLAS functionalities.

Prerequisites

1. Python is installed on target machine
2. Make version 4.x or above is installed

Build Steps

1. git clone https://github.com/amd/libflame.git
2. Run the configure script. An example below shows a few sample options. Enable/disable the other flags as per the requirement.

   With GCC (default)

   ```
   $ ./configure --enable-lapack2flame --enable-external-lapack-interfaces --enable-dynamic-build --enable-max-arg-list-hack --prefix=<your-install-dir>
   ```

   With AOCC

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

$ ./configure --enable-lapack2flame --enable-external-lapack-interfaces --
enable-dynamic-build --enable-max-arg-list-hack --enable-f2c-dotc --
prefix=<your-install-dir>
```

3. Make and install. By default, without 'prefix' configure option, the library will be installed to $HOME/flame.

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

5.1.2. Using Pre-built Binaries

AMD optimized libFLAME library binaries for Linux can be found in the following URLs:

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

Also, libFLAME binary can be installed from the AOCL master installer tar file available in the following URL. The tar file includes pre-built binaries of the other AMD libraries BLIS, LibM, FFTW, ScaLAPACK, RNG, and AMD Secure RNG.

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

5.2. Usage

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

To execute the tests, navigate to the libFLAME source directory.

```
$ make check LIBBLAS=<Full path-to-BLIS-library including the library>

Example
Using Single thread BLIS
$ make check LIBBLAS=/home/user/aocl/amd/3.x/libs/libblis.a

Using Multi-thread BLIS
$ make check LIBBLAS="/home/user/aocl/amd/3.x/libs/libblis-mt.a -lomp"
```

Execute libFLAME C++ Template API tests as follows:

Using GCC

```
$ make checkcpp LIBBLAS_PATH=<Full path-to-BLIS-library, excluding library
name>

Example:
$ make checkcpp LIBBLAS_PATH=/home/user/aocl/amd/3.x/lib
```

Using AOCC

```
$ make checkcpp LIBBLAS_PATH=<Full path-to-BLIS-library excluding library
name> LDFLAGS="-no-pie -lpthread"
```

Example

```
$ make checkcpp LIBBLAS_PATH=/home/user/aocl/amd/3.x/lib LDFLAGS="-no-pie -
lpthread"
```

Use by Applications

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

Example

With Static Library

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

With Dynamic Library

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

6. FFTW Library for AMD

AMD optimized version of 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™ processor. It is an open-source implementation of the Fast Fourier transform algorithm. It can compute transforms of real and complex valued arrays of arbitrary size and dimension.

6.1. Installation

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

6.1.1. Build FFTW from Source

Complete the following steps to build AMD Optimized FFTW for AMD EPYC processor based on Naples, Rome, and future generation architectures:

1. Download the latest stable release of AMD Optimized FFTW from the following URL: 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 AMD EPYC processors. 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-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-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-
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-
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 CPU architecture values, such as auto, znver1, znver2, or znver3 for AMD EPYC processors. The znver3 option is only supported in the AOCC compiler for now and not supported in the GCC compiler.

3. To build the library, type
   $ make
4. To install the library in preferred installation path, type
   $ make install
5. To verify the installed library, type
   $ make check

6.1.2. Using Pre-built Binaries

AMD optimized FFTW library binaries for Linux are available in the following URL:
https://developer.amd.com/amd-aocl/fftw/

AMD Optimized FFTW binary can also be installed from the AOCL master installer tar file available in the following link: https://developer.amd.com/amd-aocl/

The tar file includes pre-built binaries of other AMD Libraries BLIS, libFLAME, ScaLAPACK, LibM, aocl-sparse, RNG, and AMD Secure RNG.
Note: The pre-built libraries are prepared on a specific platform having dependencies related to OS, GCC, MPI, 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 FFTW APIs and performance benchmarking can be found under the tests/ and mpi/ directories of the FFTW source.

- Sample programs for single-threaded and multi-threaded FFTW

$ cd fftw/tests
//To run single-threaded test, type:
$ 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 the section 15.4.
//To run multi-threaded test, type:
$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 15.4.

- Sample programs for MPI FFTW

$ cd fftw/mpi
$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 section 15.4.

Additional options:
-owisdom
    On startup, read wisdom from a file wis.dat in the current directory (if it exists).
    On completion, write accumulated wisdom to wis.dat (overwriting if file exists).
    This by-passes the planner next time onwards and directly executes the read plan from wisdom.
--verify <problem>
    Verify that FFTW is computing correctly. Does not output anything unless an error.
-v<n>
    Set verbosity to <n>, or 1 if <n> is omitted. -v2 will output the created plans.

To display the AOCL version number of FFTW library, application must call the following FFTW API:
fftw_aoclversion()
The test bench executables of FFTW support the display of AOCL version using the `--info-all` option.

7. AMD LibM

AMD 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. AMD LibM is a C library which you can link into your applications to replace the compiler provided math functions. Applications can link into AMD LibM library and invoke math functions instead of compiler math functions for better accuracy and performance.

The latest AMD 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 w.r.t 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.
- Denormal inputs may produce unpredictable results. It is therefore the responsibility of the caller of these routines to ensure that their arguments are suitable.
- Also, some of the vector variants may not set appropriate IEEE error codes in the FPU.
- The vector routines will have to 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`, and `__m256d` types and 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>

v – vector
r – real
a - Array
<type> - 's' for single precision, 'd' for double precision
<vec_size> - 2 or 4 for 2 element or 4 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 vrs4_expf(__m128 x)
```

The list of available vector functions is given below. All the functions have an 'amd_' prefix and is omitted from the list to reduce the length.

Exponential

```
 * vrs4_expf, vrs4_exp2f, vrs4_exp10f, vrs4_expm1f
 * vrsa_expf, vrsa_exp2f, vrsa_exp10f, vrsa_expm1f
 * vrd2_exp, vrd2_exp2, vrd2_exp10, vrd2_expm1, vrd4_exp, vrd4_exp2
 * vrda_exp, vrda_exp2, vrda_exp10, vrda_expm1
```

Logarithmic

* vrs4_logf, vrs4_log2f, vrs4_log10f, vrs4_log1pf
* vrsa_logf, vrsa_log2f, vrsa_log10f, vrsa_log1pf
* vrd2_log, vrd2_log2, vrd2_log10, vrd2_log1p, vrd4_log
* vrda_log, vrda_log2, vrda_log10, vrda_log1p

Trigonometric

* vrs4_cosf, vrs4_sinf
* vrsa_cosf, vrsa_sinf
* vrd2_cos, vrd2_sin, vrd2_cosh, vrd2_sincos
* vrda_cos, vrda_sin

Hyperbolic

* vrs4_coshf, vrs4_tanhf
* vrs8_coshf, vrs8_tanhf

Power

* vrs4_cbrtf, vrd2_cbrt, vrs4_powf, vrd2_pow, vrd4_pow
* vrsa_cbrtf, vrda_cbrt, vrsa_powf

The scalar functions listed below are present in the library. They can be called by a standard C99 function call and naming convention and must be linked with AMD 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 AMD LibM:

Trigonometric

* cosf, cos, sinf, sin, tanf, tan, sincosf, sincos
* acosf, acos, asinf, asin, atanf, atan, atan2f, atan2

Hyperbolic

* coshf, cosh, sinhf, sinh, tanhf, tanh
* acoshf, acosh, asinhf, asinh, atanhf, atanh

Exponential & Logarithmic

* expf, exp, exp2f, exp2, exp10f, exp10, expm1f, expm1

* logf, log, log10f, log10, log2f, log2, log1pf, log1p
* logbf, logb, ilogbf, ilogb
* modff, modf, frexpf, frexp, ldexpf, ldexp
* scalbnf, scalbn, scalblnf, scalbln

Power & Absolute value

* powf, pow, fastpow, cbrtf, cbrt, sqrtf, sqrt, hypotf, hypot
* fabsf, fabs

Nearest integer

* ceilf, ceil, floorf, floor, truncf, trunc
* rintf, rint, roundf, round, nearbyintf, nearbyint
* lrintf, lrint, llrintf, llrint
* lroundf, lround, llroundf, llround

Remainder

* fmodf, fmod, remainderf, remainder

Manipulation

* copysignf, copysign, nanf, nan, finitef, finite
* nextafterf, nextafter, nexttowardf, nexttoward

Maximum, Minimum & Difference

* fdimf, fdim, fmaxf, fmax, fminf, fmin

7.1. Installation

AMD LibM binary for Linux can be found in 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 in the following link. The tar file includes pre-built binaries of other AMD Libraries BLIS, libFLAME, FFTW, RNG, and AMD Secure RNG.

7.2. Compiling AMD LibM

Minimum software requirements for compilation:

1. Gcc 9.2 or higher
2. Glibc 2.29 or higher
3. Virtualenv with python3
4. Scons 3.0.5

Compilation steps:

1. Download source from https://github.com/amd/aocl-libm-ose from the branch AOCL-3.0
2. Navigate to the folder: cd aocl-libm-ose

3. Create a virtualenvironment: virtualenv –p python3 .venv3;
4. Activate it: source .venv3/bin/activate;
5. Install scons: pip install scons;
6. Compile AMD LibM:
    a. scons –j32 install –compiler=gcc (choose parameter ‘‘aocc” for AOCC)
    b. Additional parameters: --prefix=<path to install> CC=<gcc/clang exe path> CXX=<g++ clang++ path>
    c. Verbosity options: --verbose=1
    d. Debug mode build: --debug_mode=libs
7. The libraries (static and dynamic) will be compiled and generated here: aocl-libm-ose/build/aocl-release/src/

7.3. Usage

To use AMD 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 might sometimes have a dependency on the system math library. When linking AMD 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. Whereas with the g++ compiler (for C++), it is not needed.

```
$ export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:<Path to libalm.so>

$ clang–march=native -Wall -std=c99 myprogram.c -o myprogram –L<Path to libalm.so> -lalm -lm
$ gcc –march=native -Wall -std=c99 myprogram.c -o myprogram –L<Path to libalm.so> -lalm -lm
```

For the vector calls, you must depend on compiler flag `‘-ffastmath.`
However, 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 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;
}
```

```
$ 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
```

8. AMD Optimized memcpy

AMD optimized memcpy is derived out of glibc 2.31 memcpy. This variant of memcpy brings in performance gains for memory copies of data sizes 1MB and above on the AMD Zen architecture CPUs. The source code of this variant of memcpy is released. It can be compiled and linked with the application which will use memcpy of glibc.

8.1. Building AMD Optimized memcpy

Download the source of optimized memcpy by installing the AOCL master installer, *aocl-linux-<compiler>-<version>.tar.gz*, from following URL:

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

1. After installing AOCL master installer, check for `amd-memcpy` in the root directory.
2. The source is available under `amd-memcpy/src/memcpy.c`.
3. Build the source as a shared library:

   GCC Compiler
   $ gcc -c -Wall -Werror -fpic **amd_memcpy.c**
   $ gcc -shared -o libamd_memcpy.so **amd_memcpy.o**

   (or)

   AOCC Compiler
   $ clang -c -Wall -Werror -fpic **amd_memcpy.c**
   $ clang -shared -o libamd_memcpy.so **amd_memcpy.o**

8.2. Building an Application

Any application using “memcpy” of glibc can link the libamd_memcpy.so library completing the following steps:

$ export LD_LIBRARY_PATH=<path to library>:$LD_LIBRARY_PATH

GCC Compiler:
$ gcc -L<path to library> -Wall -o <exe> <app source files> **-lamd_memcpy**
(or)
AOCC Compiler:
$ clang -L<path to library> -Wall -o <exe> <app source files> **-lamd_memcpy**

8.3. Running the Application

Run the application by setting the pre-loader environment variable.

$ LD_PRELOAD=.<path_to_library>/libamd_memcpy.so <exe> –args

9. ScaLAPACK library for AMD

ScaLAPACK is a library of high-performance linear algebra routines for parallelly distributed memory machines. It depends on the external libraries including BLAS and LAPACK for Linear Algebra computations. AMD optimized version of ScaLAPACK enables the using of BLIS and libFLAME library that have optimized dense matrix functions and solvers for the AMD EPYC™ processor family CPUs.

9.1. Installation

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

9.1.1.1. Build ScaLAPACK from source

Github link: https://github.com/amd/scalapack

Prerequisites

Building AMD optimized ScaLAPCK library requires linking to the following libraries installed using pre-built binaries or built from source:

- BLIS
- libFLAME
- A MPI library. In our experiments, we have validated with OpenMPI library

1. git clone https://github.com/amd/scalapack.git
2. $ cd scalapack
3. CMake
   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.

Compiler	Library Type	Threading	Command
GCC	Static	Single-Thread BLIS	$ cmake .. -DBUILD_SHARED_LIBS=OFF -DBLAS_LIBRARIES="<path to 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 -DUSE_F2C=ON
		Multi-thread BLIS	$ cmake .. -DBUILD_SHARED_LIBS=OFF -DBLAS_LIBRARIES="**-fopenmp** <path to BLIS library>/**libblis-mt.a**" -DLAPACK_LIBRARIES="<path to libflame library>/libflame.a" -DCMAKE_C_COMPILER=mpicc -

Compiler	Library Type	Threading	Command
			DCMAKE_Fortran_COMPILER=mpif90 -DUSE_OPTIMIZED_LAPACK_BLAS=OFF -DUSE_F2C=ON
	Shared	Single-Thread BLIS	$ cmake .. -DBUILD_SHARED_LIBS=**ON** -DBLAS_LIBRARIES="<path to 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 -DUSE_F2C=ON
		Multi-thread BLIS	$ cmake .. -DBUILD_SHARED_LIBS=**ON** -DBLAS_LIBRARIES="**-fopenmp** <path to 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 -DUSE_F2C=ON
AOCC	Static	Single-Thread BLIS	$ cmake .. -DBUILD_SHARED_LIBS=OFF -DBLAS_LIBRARIES="<path to 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 -DUSE_F2C=ON
		Multi-thread BLIS	$ cmake .. -DBUILD_SHARED_LIBS=OFF -DBLAS_LIBRARIES="**-fopenmp** <path to 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 -DUSE_F2C=ON
	Shared	Single-Thread BLIS	$ cmake .. -DBUILD_SHARED_LIBS=**ON** -DBLAS_LIBRARIES="<path to BLIS library>/libblis.a" -DLAPACK_LIBRARIES="<path

Compiler	Library Type	Threading	Command
			to libflame library>/libflame.a" -DCMAKE_C_COMPILER=mpicc -DCMAKE_Fortran_COMPILER=mpif90 -DUSE_OPTIMIZED_LAPACK_BLAS=OFF -DUSE_F2C=ON
		Multi-thread BLIS	$ cmake .. -DBUILD_SHARED_LIBS=ON -DBLAS_LIBRARIES="**-fopenmp** <path to 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 -DUSE_F2C=ON

d. Ensure CMake locates libFLAME and BLIS libraries. On completion, a message, "**LAPACK routine dgesv is found: 1**" similar to the following in CMake output will be 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 ScaLAPACK library will be created under the lib directory.

9.1.2. Using Pre-built Binaries

AMD optimized ScaLAPACK library binaries for Linux are available in the following URLs.
https://github.com/amd/scalapack/releases
https://developer.amd.com/amd-aocl/scalapack/

Also, AMD optimized ScaLAPACK binary can be installed from the AOCL master installer tar file available in the following link. The tar file includes pre-built binaries of other AMD Libraries BLIS, libFLAME, FFTW, LibM, aocl-sparse, RNG, and AMD Secure RNG.

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

9.2. Usage

The applications demonstrating the usage of ScaLAPACK APIs can be found in the TESTING directory of ScaLAPACK source package.

```
$ cd scalapack/TESTING
```

10. AMD Random Number Generator

The AMD Random Number Generator 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 five 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.

10.1. Installation

The AMD Random Number Generator binary for Linux is available in the following URL:

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

Also, the RNG binary can be installed from the AOCL master installer tar file available in the following link. The tar file includes pre-built binaries of other AMD Libraries BLIS, libFLAME, LibM, ScaLAPACK, FFTW, aocl-sparse, and AMD Secure RNG

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

10.2. Usage

To use the AMD Random Number Generator library in your application, you must link the library while building the application.

The following is a sample Makefile for an application that uses the AMD Random Number Generator library:

```
RNGDIR := <path-to-Random-Number-Generator-library>

CC := gcc

CFLAGS := -I$(RNGDIR)/include

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
```

Refer to the examples directory under the AMD Random Number Generator library install location for illustration.

11. AMD Secure RNG

The AMD Secure RNG is a library that provides the APIs to access the cryptographically secure random numbers generated by the AMD hardware based random number generator. 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. 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.

11.1. Installation

AMD Secure RNG library can be downloaded from following URL:

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

Also, AMD Secure RNG can be installed from the AOCL master installer tar file available in the following link. The tar file includes pre-built binaries of other AMD Libraries BLIS, libFLAME, LibM, ScaLAPACK, FFTW, aocl-sparse, and AMD RNG library.

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

11.2. Usage

The following source files are included in the AMD Secure RNG package:

include/secrng.h	Header file that has declaration of all the library APIs.
src_lib/secrng.c	Has 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 application

Application developers can use the included makefile to compile the source files and generate dynamic and static libraries. They can then link it to their application and invoke the required APIs.

The following code snippet shows a sample usage of the library API. In this 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.

```
//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");
}
```

12. AOCL-Sparse

aocl-sparse is a library that contains basic linear algebra subroutines for the sparse matrices and vectors optimized for AMD EPYC™ family of processors. It is designed to be used with C and C++.

The current functionality of aocl-sparse is organized in the following categories:

- Sparse Level 2 functions describe the operations between a matrix in sparse format and a vector in dense format.
- 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 2 Functions
  - 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)
- Sparse Auxiliary Functions
  - 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
- Conversion Functions
  - 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)

For more details on the aocl-sparse APIs, refer aocl-sparse API Guide in the source directory (under GitHub repo https://github.com/amd/aocl-sparse).

12.1. Installation

12.1.1. Build aocl-sparse From Source

Complete the following instructions to build aocl-sparse from source. Also, the following compile-time dependencies must be met:

- git

- CMake 3.5 or later

Download aocl-sparse

Download the latest release of aocl-sparse from the URL
https://github.com/amd/aocl-sparse

1. git clone https://github.com/amd/aocl-sparse.git
2. cd aocl-sparse

Build aocl-sparse

Complete the following steps to build different packages of the library, including dependencies and test application. aocl-sparse can be built using the following commands:

1. Create and change to the build directory.
   ```
   $ mkdir -p build/release; cd build/release
   ```
2. Run CMake as per the compiler and library type required.

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

3. Additionally, the following CMAKE options can also be used:
   a. Use `-DCMAKE_INSTALL_PREFIX=<path>` to choose the custom path. The default install path is /opt/aoclsparse/.

b. Use `-DBUILD_CLIENTS_BENCHMARKS=ON` to build the test application along with the aocl-sparse library. This is OFF by default.

4. Compile the aocl-sparse library
   ```
   $ make -j$(nproc)
   ```
5. Install aocl-sparse to /opt/aoclsparse or Custom Path
   ```
   $ make install
   ```

12.1.2. Simple Test

You can test the installation by running one of the aocl-sparse examples, after compiling the library with benchmarks.

```
# Navigate to Test binary directory
$ cd aocl-sparse/build/release/tests/staging

# Ensure that shared library is available in the library load path
$ export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:"path/to/libaoclsparse.so"

# Execute aocl-sparse example by running CSR-SPMV on randomly generated
matrix
$ ./aoclsparse-bench --function=csrmv --precision=d --sizem=1000 --sizen=1000
--sizennz=4000 --verify=1
```

12.1.3. Using Pre-built Libraries

AMD optimized aocl-sparse library binaries for Linux can be found in the following URL:

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

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

Also, aocl-sparse binary can be installed from the AOCL master installer tar file available in the following URL:
https://developer.amd.com/amd-aocl/

The tar file includes pre-built binaries of other AMD Libraries BLIS, libFLAME, LibM, FFTW, ScaLAPACK, RNG, and AMD Secure RNG.

12.2. Usage

Sample programs demonstrating the usage of aocl-sparse APIs and performance benchmarking can be found under the tests directory of aocl-sparse source.

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

Use by Applications

To use aocl-sparse in your application, you must link the library while building the application.

Example:

With Static Library

```
g++ sample_csrmv.cpp -I<path-to-aocl-sparse-header>  <path-toaocl-sparse-
library>/libaoclsparse.a -o test_aoclsparse.x
```

With Dynamic Library

```
g++ sample_csrmv.cpp -I<path-to- aocl-sparse-header>  <path-to aocl-sparse-
library>/libaoclsparse.so -o test_aoclsparse.x
```

The following is a sample cpp file depicting the usage of dcsrmv API of aocl-sparse:

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

int main(int argc, char* argv[])
{
   aoclsparse_int       M       = 5;
   aoclsparse_int       N       = 5;
   aoclsparse_int       nnz     = 8;
   aoclsparse_operation   trans    = aoclsparse_operation_none;

   double alpha = 1.0;
   double beta  = 0.0;

   // Print aoclsparse version
   aoclsparse_int ver;
   aoclsparse_get_version(&ver);
   std::cout << "aocl-sparse version: " << ver / 100000 << "." << ver / 100 % 1000 << "."
          << ver % 100 << 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);

   // Initialise matrix
   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 , 6 , 1.050e+01, 1.500e-02, 2.505e+02, -2.800e+02 , 3.332e+01 ,
1.200e+01};
   // Initialise vectors
   double x[N] = { 1.0, 2.0, 3.0, 4.0, 5.0};
   double y[M];

   std::cout << "Invoking aoclsparse_dcsrmv..";
   //Invoke SPMV API for CSR storage format(double precision)
   aoclsparse_dcsrmv(trans,
                &alpha,
```

```
                M,
                N,
                nnz,
                csr_val,
                csr_col_ind,
                csr_row_ptr,
                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);
    return 0;
}
```

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

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

13. AOCL Spack Recipes

Spack is a package manager for supercomputers, Linux, and macOS. It makes installing scientific software easy. With Spack, you can build a package with multiple versions, configurations, platforms, and compilers; and all these builds can co-exist on the same machine.

Note:

- From AOCL 2.2 release, the Spack recipes for the AMD optimized libraries of BLIS, libFLAME, ScaLAPACK, and FFTW will be available in the new GitHub repository https://github.com/amd/spack. The earlier AMD Spack GitHub repository https://github.com/amd/aocl-spack is deprecated.
- AOCL Spack recipes for BLIS, libFLAME, ScaLAPACK, and FFTW libraries are also upstreamed in main community repository https://github.com/spack/spack.

13.1. AOCL Spack Environment Setup

1. Clone the AMD Spack GitHub repository.

```
$ git clone https://github.com/amd/spack.git
```

2. Set environment path for the Spack shell.

```
$ export SPACK_ROOT=/path/to/spack
$ source $SPACK_ROOT/share/spack/setup-env.csh
```

13.2. Install AOCL packages

The Spack recipes for AMD optimized libraries of BLIS, libFLAME, ScaLAPACK, and FFTW are available in GitHub repository https://github.com/amd/spack.

13.2.1. Install amdblis Spack Package

Build and install sequential BLIS

```
$ spack install amdblis
```

Build and install BLIS with OpenMP multithreading

```
$ spack install amdblis threads=openmp
```

13.2.2. Install amdlibflame Spack Package

```
$ spack install amdlibflame
```

13.2.3. Install amdfftw Spack Package

```
$ spack install amdfftw
```

13.2.4. Install amdscalapack Spack Package

Install AMD ScaLAPACK with AMD BLIS and libFLAME libraries.

```
$ spack install amdscalapack ^amdlibflame ^amdblis
```

13.2.5. Install legacy AOCL versions

By default spack installs the latest versions of the AOCL libraries. however, one can install legacy versions of the libraries by using suffix '@' followed by the desired legacy version.

For example, to install 2.2 version BLIS, run following command

```
$ spack install amdblis@2.2
```

13.3. Useful Spack Commands

Few useful Spack commands to get additional information on the Spack packages are as follows. For more information, refer the Spack documentation.

Display BLIS package info and supported versions	`$ spack info amdblis`
Install BLIS	`$ spack install amdblis`
Verify installed contents	`$ spack spec amdblis`
Go to BLIS install-directory	`$ spack cd -i amdblis`
Install other versions of amdblis package, use @<version-number>	`$ spack install -v amdblis@2.1`
Check supported versions, run the command	`$ spack versions amdblis`
Build and install BLIS 2.2 with OpenMP multithreading	`$ spack install amdblis@2.2 threads=openmp`

Under the BLIS installation directory, a *.spack* directory containing the following files or directories will be available:

. spack-build-env.txt	Captures build environment details
. spack-build-out.txt	Captures build output
. spec.yaml	Captures installed version, arch, compiler, namespace, configure parameters and package hash value
. repos	Directory containing spack recipe and repo namespace files

13.4. Uninstall AOCL Packages

Following are sample list of commands for unstalling AOCL Spack packages

Uninstall BLIS default package	`$ spack uninstall amdblis`
Uninstall libFLAME default package	`$ spack uninstall amdlibflame`
Uninstall FFTW default package	`$ spack uninstall amdfftw`
Uninstall BLIS based out of different versions	`$ spack uninstall amdblis@2.0`
Uninstall BLIS based out of hash values	`$ spack uninstall amdblis/43reafx`

14. Applications integrated to AOCL

This section provides examples on how AOCL can be linked with some of the important High Performance Computing (HPC) and CPUintensive applications and libraries.

14.1. High-performance LINPACK Benchmark (HPL)

HPL [3] is a software package that solves a (random) dense linear system in double precision (64 bits) arithmetic on distributed-memory computers. HPL is a LINPACK benchmark which measures the floating-point rate of execution for solving a linear system of equations.

For building a HPL binary from the source code, you must edit the MPxxx and LAxxx directories in your architecture-specific Makefile to match the installed locations of your MPI and Linear Algebra library. For BLIS, use the F77 interface with F2CDEFS = -DAdd__ -DF77_INTEGER=int -DStringSunStyle. Use cthe multi-threaded BLIS for an optimal performance.

Setup HPL.dat before running the benchmark.

Configuring HPL.dat

HPL.dat file contains 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.

Running the Benchmark

Combination of multithreading (through OpenMP library) and MPI demonstrates and optimal performance. Set the number of MPI tasks to number of L3 caches in the system for a good performance.

HPL benchmark is found to generate better single node performance number with following recipes:

1) AMD ROME (EPYC 7742):

Dual socket AMD EPYC 7742 system consists of 32 CCX and all of them have a L3 cache and total 2 x 64 cores. 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.

```
export BLIS_IC_NT=4

export BLIS_JC_NT=1
```

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 further improve the performance.

2) AMD MILAN

The number of MPI ranks and maximum thread count per MPI rank depends on the specific EPYC SKU. For a better performance, set the number of MPI tasks to the number of CCX and bind each MPI task to the L3 cache.

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. AOCL Tuning Guidelines

This section provides tuning recommendations for AOCL to derive an optimal performance on AMD EPYC™ and future generation architecture processors.

15.1. BLIS DGEMM Multi-thread Tuning

A BLIS library can be used on multiple platforms and applications. Multi-threading adds more configuration options at the runtime. This section explains some of the parameters that can be tuned to get best performance for the your needs.

15.1.1. Library Usage Scenarios

The application and library are single-threads

This is straight forward - no special instructions needed. You can *export BLIS_NUM_THREADS=1* indicating you are running BLIS in a single-thread mode. To that BLIS runs in a single-thread mode you must not even set environment variable *OMP_NUM_THREADS*.

The application is single-thread and the library is multi-thread

You can either use *OMP_NUM_THREADS* or *BLIS_NUM_THREADS* to define number of threads for the library. However, it is recommend that you use *BLIS_NUM_THREADS*.

Example:

```
$ export BLIS_NUM_THREADS=128 // Here BLIS runs at 128 threads.
```

Apart from setting the number of threads, you must pin the threads to the cores. This can be done using GOMP_CPU_AFFINITY or numactl as shown below

```
$ BLIS_NUM_THREADS=8 GOMP_CPU_AFFINITY=0-7 <./application>
```

Or

```
$ BLIS_NUM_THREADS=16 GOMP_CPU_AFFINITY=0-15 numactl --i=all <./application>
```

```
$ BLIS_NUM_THREADS=16 numactl -C 0-15 --interleave=all <./test_application.x>
```

However, it is recommended that you use *GOMP_CPU_AFFINITY*, because a better performance is expected with this type of thread binding.

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, for 32 threads, if you run DGEMM without *OMP_PROC_BIND* settings, the performance will be less. However, if you start using

OMP_PROC_BIND=true, performance improves greatly. This problem is not noticed with libgomp, using gcc compiler. For the gcc compiler, the processor affinity defined using numactl is enough.

15.1.2. The application is multi-thread and the library are single-threads

When the application is running multi-thread and number of threads are set using *OMP_NUM_THREADS*, then it is mandatory to set *BLIS_NUM_THREADS* to one. Otherwise, BLIS will run in multithread mode with the number of threads equal to *OMP_NUM_THREADS*. This may result in a poor performance. For example, if the application is launching 64 threads, each thread will call the BLIS library function. Launching 64 * 64 threads, each application thread launches 64 child threads from the BLIS library call. To avoid this, always set *BLIS_NUM_THREADS = 1*.

15.1.3. Both the Application and the Library are multi-threads

This is a typical scenario of nested parallelism. To individually control the threading at application and at the 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 will be 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 BLIS-MT.

15.1.4. Architecture Specific Tuning

AMD Rome and Milan

To achieve the best DGEMM multi-thread performance on AMD Rome and Milan processors, follow the below steps:

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
```

AMD Naples

To achieve the best DGEMM multi-thread performance on the AMD Naples processors, follow the below steps.

The header file, *bli_family_zen.h* located under BLIS source directory *blis**config**zen* defines certain macros that help control the block sizes used by 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 BLIS with multi-thread option as mention in the section Multi-threaded BLIS.
3. Link the generated 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 BLIS with multithread option as mentioned in the section Multi-threaded BLIS.
3. Link the generated 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
   ```

15.2. BLIS DGEMM Block-size Tuning for Single and Multi-instance mode

BLIS DGEMM performance is largely impacted by the block sizes used by 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 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 under AMD BLIS GitHub repo is set to extract best performance for such HPC applications/benchmarks which use single-threaded BLIS and run in multi-instance mode on AMD EPYC "Zen" core processors. However, if your application runs as a single instance, the block sizes for optimal performance will vary.

The following settings will help you choose the optimal values for the block sizes based on the way the application is run:

AMD Rome

1. Open the file *bli_family_zen2.h* under the BLIS source.
2. $ cd "*config/zen2/ bli_family_zen2.h*"For applications/benchmarks running in multi-instance mode and using multi-threaded BLIS, ensure that the macro, AOCL_BLIS_MULTIINSTANCE is set to 0. As of AMD BLIS 2.x release, this is the default setting. HPL benchmark is found to generate better performance numbers using this setting for multi-threaded BLIS.

```
#define AOCL_BLIS_MULTIINSTANCE       0
```

AMD Naples

1. Open the file *bli_cntx_init_zen.c* under the 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
```

3. **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 AMD BLIS 2.x release, this is the default setting.

```
#define BLIS_ENABLE_SINGLE_INSTANCE_BLOCK_SIZES       0
```

   The optimal block sizes for this mode on AMD EPYC™ are defined in the file *config/zen/bli_cntx_init_zen.c*

```
    bli_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 );
```

4. **Single instance mode**
   For applications running as a single instance, ensure that the macro BLIS_ENABLE_SINGLE_INSTANCE_BLOCK_SIZES is set to 1.

```
#define BLIS_ENABLE_SINGLE_INSTANCE_BLOCK_SIZES       1
```

   The optimal block sizes for this mode on AMD EPYC™ are defined in the file *config/zen/bli_cntx_init_zen.c*

```
    bli_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 );
```

15.3. Performance Suggestions for Skinny Matrices

BLIS provides a selective packing for GEMM when one or two-dimensions of a matrix is exceedingly small. This feature is only available when sup handling is enabled by default.

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 Dim(A) >> Dim(B)
$BLIS_PACK_A=1 ./test_gemm_blis.x – will give better performance.
IF Dim(A) << Dim(B)
$BLIS_PACK_B=1 ./test_gemm_blis.x – will give better performance.

15.4. AMD Optimized FFTW Tuning Guidelines

The following are the tuning guidelines to get the best performance out of AMD optimized FFTW:

1. Use the configure option *--enable-amd-opt* to build the library targeted. This option enables all the improvements and optimizations meant for AMD EPYC™ CPUs.
2. 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.*
3. An optional configure option *--enable-amd-trans* is provided and it may benefit the performance of transpose operations in case of very large FFT problem sizes. This feature is to be used only when running in single-thread and single instance mode.
4. 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.
5. AMD optimized fast planner can be enabled using optional configure option *--enable-amd-fast-planner*. This option can be used to reduce the planning time without much tradeoff in the performance. It is supported for single and double precisions.
6. For best performance, use the "-opatient" planner flag of FFTW.
   A sample running of FFTW bench test application with "-opatient" 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, forward transform.
7. 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:
   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.
8. When running MPI FFTW test, set the appropriate MPI mapping, binding, and rank options.
   For example, to run 64 MPI rank FFTW on a 64-core AMD EPYC™ processor, use:
   mpirun --map-by core --rank-by core --bind-to core -np 64 ./mpi-bench -opatient -s icf65536

16. Appendix

16.1. Check AMD Server Processor Architecture

To check if your AMD CPU is of Naples, Rome, or Milan based architecture, perform the following steps on Linux:

1. Run the lscpu command:
   ```
   $ lscpu
   ```
2. Check the values of CPU family and Model fields:
   i. For Naples

cpu family : 23 model : Values in the range <1 – 47>

   ii. For Rome

cpu family : 23 model : Values in the range < 48 – 63>

   iii. For Milan

cpu family : 25 model : Values in the range < 1 – 15>

16.2. Application Notes

16.2.1. AMD Optimized FFTW

a. Quad precision is recently supported in AMD Optimized FFTW with the AOCC compiler (AMD clang version 10 onwards).

17. Technical Support and Forums

For questions and issues about AOCL, you can reach us on toolchainsupport@amd.com.

18. References

1. https://developer.amd.com/amd-aocl/
2. http://www.netlib.org/scalapack/
3. http://www.netlib.org/benchmark/hpl/
4. https://dl.acm.org/citation.cfm?id=2764454
5. https://github.com/flame/blis
6. http://fftw.org/