AMD Optimizing CPU Libraries User Guide

Version 2.2

Table of Contents

1. Introduction

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

AOCL comprise of eight packages, primarily,

1. **BLIS (BLAS Library)** – BLIS is a portable open-source software framework for instantiating high-performance Basic Linear Algebra Subprograms (BLAS) functionality.
2. **libFLAME (LAPACK)** - libFLAME is a portable library for dense matrix computations, providing much of the functionality present in 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 thereof.
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** - AMD Random Number Generator Library is a pseudorandom number generator library
7. **AMD Secure RNG** - The AMD Secure Random Number Generator (RNG) is a library that provides APIs to access the cryptographically secure random numbers generated by AMD's hardware random number generator implementation.
8. **AOCL-Sparse(New)** - Library that contains basic linear algebra subroutines for sparse matrices and vectors optimized for AMD EPYC family of processors

In addition, we provide

- Spack(https://spack.io/) based recipes for installing BLIS, libFLAME and FFTW libraries.
- AMD optimized memcpy library (New)

Latest information on the AOCL release and installers are available in the following AMD developer site. https://developer.amd.com/amd-aocl/.

For any issues or queries regarding the libraries, please contact toolchainsupport@amd.com.

AOCL 2.2 includes several performance improvements for AMD Rome based microprocessor architecture in addition to Naples architecture. Please check Appendix Check AMD Server Processor Architecture to determine underlying the 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 18.04 LTS
- CentOS 7.6
- RHEL 8.1
- SLES 15 SP3

Compilers

- GCC 7.3 and above
- AOCC 2.2 (https://developer.amd.com/amd-aocc/)

3. AOCL Installation

AOCL can be installed by one of the following mechanisms

1. **Build from Source**
   The open source libraries of AOCL suite including BLIS, libFLAME, FFTW, ScaLAPACK and aocl-sparse can be downloaded from GitHub and built from source

   BLIS: https://github.com/amd/blis
   libFLAME: https://github.com/amd/libflame
   FFTW: https://github.com/amd/amd-fftw
   ScaLAPACK: https://github.com/amd/scalapack
   aocl-sparse: https://github.com/amd/aocl-sparse

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

2. **Master installer and Tar packages of AOCL binaries**
   AOCL master installer is available in the '*Download*' section of the following link. The master installer can used to install entire AOCL library suite
   https://developer.amd.com/amd-aocl/.

   Individual library binaries can be downloaded as well from the respective libraries page.
   For example BLIS and libFLAME tar packages are available in following link
   https://developer.amd.com/amd-aocl/blas-library/

3. **Debian and RPM Packages**
   Debian and RPM packages of AOCL are available in the '*Download*' section of the following link
   https://developer.amd.com/amd-aocl/
   The package name used in following installation steps is based on 'gcc' build. The same applies for AOCC build by replacing 'gcc' with 'aocc'.

   Steps to install AOCL Debian package

   1. Download AOCL 2.2 RPM package to target machine
   2. Check install path before installing
      $ dpkg -c aocl-linux-gcc-2.2.0_1_amd64.deb
   3. Install the package. Prerequisites: User requires sudo previlage
      $ sudo dpkg -i aocl-linux-gcc-2.2.0_1_amd64.deb
      Or
      $ sudo apt install ./aocl-linux-gcc-2.2.0_1_amd64.deb
   4. Display installed package information along with package version and short description
      $ dpkg -s aocl-linux-gcc-2.2.0
   5. List contents of package
      $dpkg -L aocl-linux-gcc-2.2.0

To uninstall the Debian package

```
$ sudo dpkg -r aocl-linux-gcc-2.2.0
(or)
$ sudo apt remove aocl-linux-gcc-2.2.0
```

Steps to install AOCL RPM package

1. Download the AOCL 2.2 RPM package to target machine
2. Install the package. Prerequisites: User requires sudo previlage
   ```
   $ sudo rpm -ivh aocl-linux-gcc-2.2.0-1.x86_64.rpm
   ```
3. Display installed package information along with package version and short description
   ```
   $ rpm -qi aocl-linux-gcc-2.2.0-1.x86_64
   ```
4. List contents of package
   ```
   rpm -ql aocl-linux-gcc-2.2.0-1
   ```

Uninstall AOCL RPM package

```
$ rpm -e aocl-linux-gcc-2.2.0-1
```

4. BLIS library for AMD

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

AMD's optimized version of BLIS supports C, FORTRAN and C++ Template interfaces for BLAS functionalities.

4.1. Installation

BLIS can be installed either from source or pre-built binaries

4.1.1. Build BLIS from source

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

4.1.1.1. Single-thread BLIS

Here are the build instructions for single threaded AMD BLIS.

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

2. Depending on the target system, and build environment, one would have to enable/disable suitable configure options. The following steps provides instructions for compiling on AMD CPU core based platforms. For a complete list of options and their description, type ./configure –help.

3. Staring 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.

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

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

4. `$ make`

5. `$ make install`

The BLIS binary (libblis) included in AOCL master installer package has been built with "`zen2`" 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, one would have to enable/disable suitable configure options. The following steps provides instructions for compiling on AMD CPU core based platforms. For a complete list of options and their description, type ./configure –help.
3. Staring 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.

   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> CC=clang CXX=clang++ auto
   ```

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

2. `$ make`

3. `$ make install`

Note: For HPC Applications – disable Small Unpacked Kernel(sup) feature as shown below:

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

$ make

$ make install

The multi-thread BLIS binary (libblis-mt) included in AOCL master installer package has been built with OpenMP threading mode and "`zen2`" config. For more information on multi-threaded implementation in BLIS refer here.

4.1.2. Using pre-built binaries

AMD optimized BLIS library binaries for Linux can be found in the following links.
https://github.com/amd/blis/releases
https://developer.amd.com/amd-aocl/blas-library/

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

The master installer includes both single threaded and multi-threaded BLIS binaries. Both BLIS binaries have been built with "`zen2`" config. The multi-thread BLIS binary (libblis-mt) included in AOCL master installer package has been 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 test cases which demonstrate usage of BLIS APIs.

To execute the tests, navigate to the BLIS source directory,

```
$ make check
```

Execute BLIS C++ Template API tests as below

```
$ make checkcpp
```

Use by Applications

To use BLIS in your application, you just need to link the library while building the application

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 application developers access to BLIS implementations via traditional FORTRAN BLAS API calls, that can be used in FORTRAN as well as 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

BLIS source directory contains test suite to verify the functionality of BLIS and BLAS APIs. The test suite invokes APIs with different inputs and verify that the results are withing expected tolerance limits.

Running Test Suite:

BLIS source directory contains test suite to verify the functionality of BLIS and BLAS APIs. The test suite invokes APIs with different inputs and verify that the results are withing expected tolerance limits.

For detailed information refer to: https://github.com/flame/blis/blob/master/docs/Testsuite.md

Test suite is invoked by running following command

```
$ make test
```

Example run of the testsuite is as shown below.

```
$:~/blis$ make test
Compiling obj/zen2/testsuite/test_addm.o
Compiling obj/zen2/testsuite/test_addv.o
.
<<< More compilation output >>>
.
Compiling obj/zen2/testsuite/test_xpbym.o
Compiling obj/zen2/testsuite/test_xpbyv.o
Linking test_libblis-mt.x against 'lib/zen2/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/zen2/blastest/cblat1.o
Compiling obj/zen2/blastest/abs.o
.
<<< More compilation output >>>
.
Compiling obj/zen2/blastest/wsfe.o
Compiling obj/zen2/blastest/wsle.o
Archiving obj/zen2/blastest/libf2c.a
Linking cblat1.x against 'libf2c.a lib/zen2/libblis-mt.a  -lm -lpthread -fopenmp -
lrt'
Running cblat1.x > 'out.cblat1'
.
<<< More compilation output >>>
.
Linking zblat3.x against 'libf2c.a lib/zen2/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 also has API specific test drivers, this section explains how to use this driver for specific set of matrix sizes.

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

Follow these steps to execute the GEMM tests on specific inputs.

Enabling File Inputs:

By default, file input/output are disabled (instead it use start, end and step sizes). To enable file inputs

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

Build test driver:

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

Create Input file:

The input file expects matrix sizes and strides in following format. Each dimension is separated by space and each entry is separated by new line. (Please see example below for details)

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 ldA, ldB and ldC are
ldA >= M
ldB >= K
ldC >= M

Running the tests:

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

Example run with test driver for GEMM.

```
$ 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 FORTRAN applications through the standard BLAS API.

For example, see below, FORTRAN code that does double precision general matrix-matrix multiplication. It calls the 'DGEMM' BLAS API function to accomplish this. An example 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
```

Example compilation command, for the above code, with gfortran compiler:

```
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 one can always use the native BLIS API for the same, 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. Note that (a) the matrices are transposed to account for the row-major storage of C and the column-major convention of BLAS (inherited from FORTRAN), (b) the function arguments are passed by address, again to be in line with FORTRAN conventions, (c) there is a trailing underscore in the function name ('dgemm_'), as BLIS' BLAS APIs expect that (FORTRAN compilers add a trailing underscore), and (d) "blis.h" is included as a header. An example command to compile it and link with the BLIS library is also shown below the 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;
}
```

Example compilation command, for the above code, with gcc compiler:

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

Using BLIS with CBLAS API

The C code below shows using CBLAS APIs for the same functionality listed above. Note that (a) CBLAS Layout option allows us to choose between row-major and column-major layouts (row-major layout is used in the example, which is in line with C-style), (b) the function arguments can be passed by value also, and (c) "cblas.h" is included as a header. An example command to compile it and link with the BLIS library is also shown below the code. Also, note that, in order to get CBLAS API with BLIS, one has to supply 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;
}
```

Example compilation command, for the above code, with gcc compiler:

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

4.3. Function call tracing in BLIS

AMD BLIS library provides 2 different way to perform function call tracing, both methods are explained in sections below. Both methods need recompilation of the library on the target system.

Key Features:

1. Can be enabled/disabled at compile time.
2. When these features are disabled at compile time, they do not need any runtime resources and does not affect performance.
3. All traces are thread safe.

4.3.1. Comprehensive call tracing

Comprehensive call tracing is implemented using compiler assisted instrumentation, hence this is enabled /disabled for the complete library and application. It also uses record/replay method.

When application with the instrumented library is ran, it will create a trace file for each thread created by this process. The thread specific files will have call trace associated for all functions called by that thread.

To use this feature please go the following 3 steps in that order.

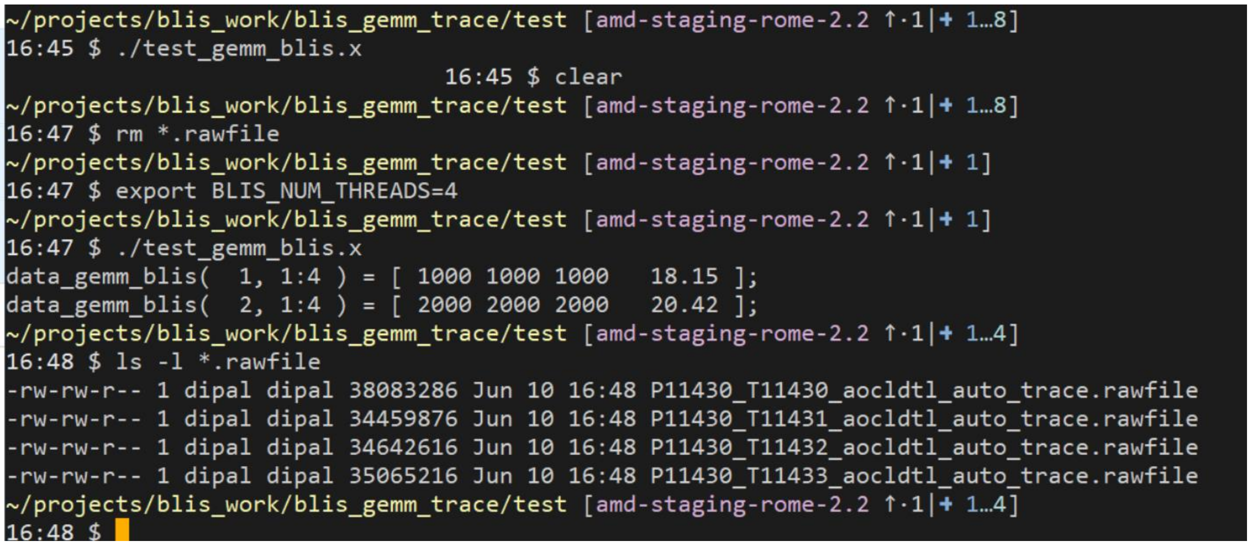
1. **Building the BLIS Library for call tracing.**
   - Building BLIS with comprehensive call trace support (This section explains steps using gcc, please refer to section 4.1.1 Build BLIS from source for how it can be built with other compilers).

   ```
   $ ./configure --enable-cblas --prefix=<your-install-dir> auto
   $ make -j ETRACE_ENABLE=1
   ```

   - Link the library to application along with additional library “-ldl”
   - For inbuild test suite please build them as give below

   ```
   $ make -j ETRACE_ENABLE=1 checkblis
   or
   $ cd test; make -j ETRACE_ENABLE=1
   ```

2. **Recoding traces:**
   - Once the application is built, just run the application as usual
   - Figure 1: Example of recoding trace data, shows example run using inbuilt test_gemm application.
   - As shown the trace data for each thread is saved in the file with following naming conventions `P<process id>_T<thread id>_aocldtl_auto_trace.rawfile`
   - Process id helps to differentiate between traces from multiple runs in the same folder.
   - These traces are not in human readable format, they need to be decoded/replayed as explained in step 3



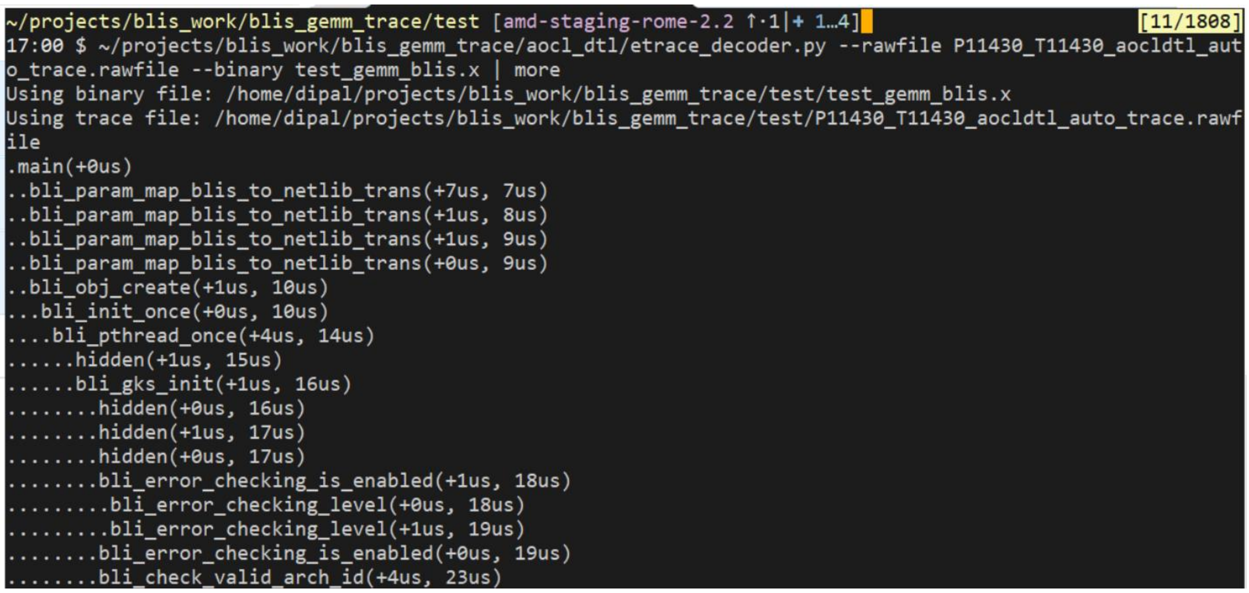
```
~/projects/blis_work/blis_gemm_trace/test [amd-staging-rome-2.2 ↑·1|+ 1…8]
16:45 $ ./test_gemm_blis.x
                                16:45 $ clear
~/projects/blis_work/blis_gemm_trace/test [amd-staging-rome-2.2 ↑·1|+ 1…8]
16:47 $ rm *.rawfile
~/projects/blis_work/blis_gemm_trace/test [amd-staging-rome-2.2 ↑·1|+ 1]
16:47 $ export BLIS_NUM_THREADS=4
~/projects/blis_work/blis_gemm_trace/test [amd-staging-rome-2.2 ↑·1|+ 1]
16:47 $ ./test_gemm_blis.x
data_gemm_blis(  1, 1:4 ) = [ 1000 1000 1000   18.15 ];
data_gemm_blis(  2, 1:4 ) = [ 2000 2000 2000   20.42 ];
~/projects/blis_work/blis_gemm_trace/test [amd-staging-rome-2.2 ↑·1|+ 1…4]
16:48 $ ls -l *.rawfile
-rw-rw-r-- 1 dipal dipal 38083286 Jun 10 16:48 P11430_T11430_aocldtl_auto_trace.rawfile
-rw-rw-r-- 1 dipal dipal 34459876 Jun 10 16:48 P11430_T11431_aocldtl_auto_trace.rawfile
-rw-rw-r-- 1 dipal dipal 34642616 Jun 10 16:48 P11430_T11432_aocldtl_auto_trace.rawfile
-rw-rw-r-- 1 dipal dipal 35065216 Jun 10 16:48 P11430_T11433_aocldtl_auto_trace.rawfile
~/projects/blis_work/blis_gemm_trace/test [amd-staging-rome-2.2 ↑·1|+ 1…4]
16:48 $
```

Figure 1: Example of recoding trace data

3. **Replay trace data:**
   - The python script required to replay the traces is available in <blis root folder>/aocl_dtl
   - Replay operation can be performed using following command line.

```
~/blis/aocl_dtl/etrace_decoder.py
--rawfile ~/blis/test/P11430_T11430_aocldtl_auto_trace.rawfile
--binary ~/blis/test/test_gemm_blis.x
```

- The decode script will warn the user if there is timestamp mismatch and if it is not able to find the function name for all symbols.
- Figure 2: Example of replaying trace data, show the example run for test_gemm application trace replay
- The dots (.) before the function names represent call stack level. For each function time elapsed from the previous entry and from the beginning are shown.



```
~/projects/blis_work/blis_gemm_trace/test [amd-staging-rome-2.2 ↑·1|+ 1…4]                    [11/1808]
17:00 $ ~/projects/blis_work/blis_gemm_trace/aocl_dtl/etrace_decoder.py --rawfile P11430_T11430_aocldtl_aut
o_trace.rawfile --binary test_gemm_blis.x | more
Using binary file: /home/dipal/projects/blis_work/blis_gemm_trace/test/test_gemm_blis.x
Using trace file: /home/dipal/projects/blis_work/blis_gemm_trace/test/P11430_T11430_aocldtl_auto_trace.rawf
ile
.main(+0us)
..bli_param_map_blis_to_netlib_trans(+7us, 7us)
..bli_param_map_blis_to_netlib_trans(+1us, 8us)
..bli_param_map_blis_to_netlib_trans(+1us, 9us)
..bli_param_map_blis_to_netlib_trans(+0us, 9us)
..bli_obj_create(+1us, 10us)
...bli_init_once(+0us, 10us)
....bli_pthread_once(+4us, 14us)
......hidden(+1us, 15us)
......bli_gks_init(+1us, 16us)
........hidden(+0us, 16us)
........hidden(+1us, 17us)
........hidden(+0us, 17us)
........bli_error_checking_is_enabled(+1us, 18us)
.........bli_error_checking_level(+0us, 18us)
.........bli_error_checking_level(+1us, 19us)
........bli_error_checking_is_enabled(+0us, 19us)
........bli_check_valid_arch_id(+4us, 23us)
```

Figure 2: Example of replaying trace data

Usages & Limitations:

1. Since this method captures trace information for all the function, it can be used to identify the performance hotspot associated with give application.
2. When tracing is enabled, performance will see significant drop.
3. This method is quite resource heavy, it can generate huge amount of data. Based on hardware configuration i.e. diskspace, number of cores & threads needed for execution it may result in sluggish or non-responsive system.
4. It is recommended to narrow down application requirements to the area of interest so that number of inputs and threads are minimized.

4.3.2. Selective call tracing

Selective call tracing is implemented using hard instrumentation of the BLIS code. Here functions are grouped as per there position in the call stack. User can configure the level till which traces will be generated.

Selective call tracing also needs recompilation and some source code modification to get it working, following section explains the steps needed to enable and view the traces. Unlike comprehensive tracing, selective trace generated human readable output, so no additional steps are needed to replay/decode the call trace.

1. **Enable trace support**

   - For this step user needs to modify the source code to enable selective tracing

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

   - Change following macro from 0 to 1

     ```
     #define AOCL_DTL_TRACE_ENABLE       0
     ```

2. **Configure the trace depth level.**
   - For this step user needs to modify the source code to enable selective tracing

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

   - Change following macro as needed, to begin with Level 5 should be good compromise in terms of details and resource requirement, Higher level we go dipper in the call stack and lower level reduce the depth of the call stack used for trace generation.

     ```
     #define AOCL_DTL_TRACE_LEVEL  AOCL_DTL_LEVEL_TRACE_5
     ```

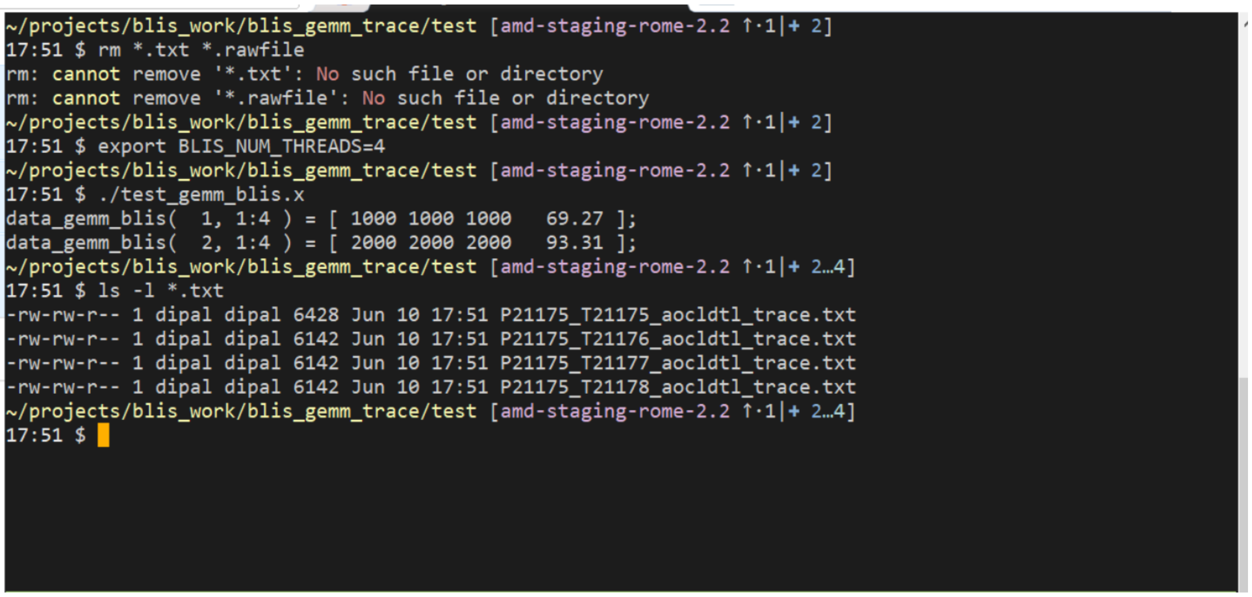
3. **Build the library**
   - Library is built normally as explain in section 4.1.1 Build BLIS from source
4. **Generate trace data**
   - Run the application normally, trace output files for each thread will be generated in the current folder.
   - Figure 3: Example run of selective call tracing, show the example of running selective call tracing using test_gemm application
   - As shown the trace data for each thread is 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_trace.txt
   ```

Usages & Limitations:

1. This method needs comparatively less resources than comprehensive call tracing. It also provides direct human readable output.
2. Biggest drawback of this method is that user can only see traces for which developers have already done instrumentation. Covering any other function in the selective trace has to be done my modifying the code.
3. This method is more suitable if area of problem is already know.
4. When tracing is enabled, performance will see significant drop.

5. EIt is recommended to narrow down application requirements to the area of interest so that number of inputs and threads are minimized.

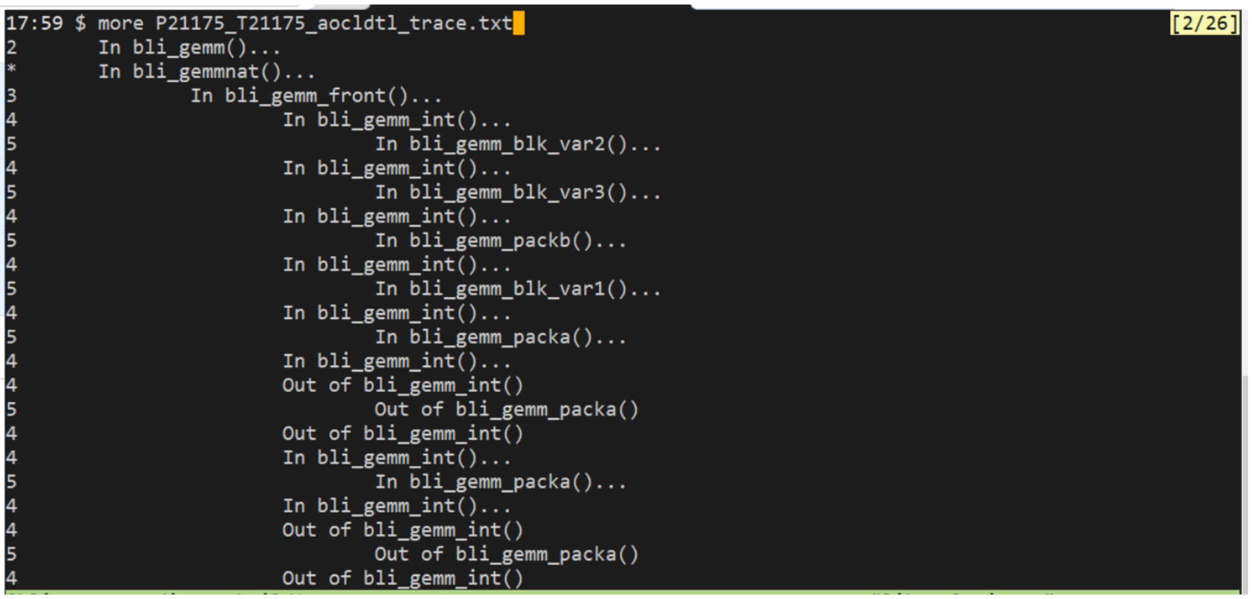


```
~/projects/blis_work/blis_gemm_trace/test [amd-staging-rome-2.2 ↑·1|✚ 2]
17:51 $ rm *.txt *.rawfile
rm: cannot remove '*.txt': No such file or directory
rm: cannot remove '*.rawfile': No such file or directory
~/projects/blis_work/blis_gemm_trace/test [amd-staging-rome-2.2 ↑·1|✚ 2]
17:51 $ export 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 3: Example run of selective call tracing

5. View trace data

- Output of selective trace is already in human readable format, just open the file in any of the text editor.
- Figure 4: Example output of selective call trace, show the example out for one of the threads of test_gemm application.
- First column show the level in call stack for the given function.
- Trace is also independent according to the position of the function in the call stack.



```
17:59 $ more P21175_T21175_aocldtl_trace.txt                              [2/26]
2       In bli_gemm()...
*       In bli_gemmnat()...
3               In bli_gemm_front()...
4                       In bli_gemm_int()...
5                               In bli_gemm_blk_var2()...
4                       In bli_gemm_int()...
5                               In bli_gemm_blk_var3()...
4                       In bli_gemm_int()...
5                               In bli_gemm_packb()...
4                       In bli_gemm_int()...
5                               In bli_gemm_blk_var1()...
4                       In bli_gemm_int()...
5                               In bli_gemm_packa()...
4                       In bli_gemm_int()...
4                       Out of bli_gemm_int()
5                               Out of bli_gemm_packa()
4                       Out of bli_gemm_int()
4                       In bli_gemm_int()...
5                               In bli_gemm_packa()...
4                       In bli_gemm_int()...
4                       Out of bli_gemm_int()
5                               Out of bli_gemm_packa()
4                       Out of bli_gemm_int()
```

Figure 4: Example output of selective call trace

5. libFLAME library for AMD

libFLAME is a portable library for dense matrix computations, providing much of the functionality present in Linear Algebra Package (LAPACK). It includes a compatibility layer, FLAPACK, which includes complete LAPACK implementation. The library provides scientific and numerical computing communities with a modern, high-performance dense linear algebra library that 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 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 BLIS library which includes optimizations for the AMD EPYC™ processor family, libFLAME enables running high performing LAPACK functionalities on AMD platform. AMD's version of libFLAME supports C, FORTRAN and C++ Template interfaces for LAPACK functionalities.

5.1. Installation

libFLAME can be installed either from 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. Applications which use libFLAME will have to link with BLIS (or other BLAS libraries) for BLAS functionalities.

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

2. Run configure script. Example below shows few sample options. Enable/disable other flags as needed

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

   With AOCC
   ```
   $ ./configure --enable-lapack2flame --enable-external-lapack-
   interfaces --enable-dynamic-build --enable-max-arg-list-hack --
   prefix=<your-install-dir> CC=clang CXX=clang++ F77=flang
   ```

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

- `$ make`
- `$ make install`

5.1.2. Using pre-built binaries

AMD optimized libFLAME library binaries for Linux can be found in the following links.

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

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

5.2. Usage

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

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

```
$ cd test
$ make LIBBLAS=<Full path-to-BLIS-library including the library>
Example
$ make LIBBLAS=/home/user/aocl/amd/2.x/libs/libblis.a
$ ./test_libflame.x
```

Run libFLAME C++ Template API tests as below

From the libFLAME source directory,

Using GCC

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

Example:
$ make checkcpp LIBBLAS_PATH=/home/user/aocl/amd/2.x/libs/libblis.a
```

Using AOCC

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

Example:

```
$ make checkcpp LIBBLAS_PATH=/home/user/aocl/amd/2.x/libs/libblis.a
LDFLAGS="-no-pie -lpthread"
```

6. FFTW library for AMD

AMD's 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-values arrays of arbitrary size and dimension.

6.1. Installation

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

6.1.1. Build FFTW from source

Here are the steps to build AMD Optimized FFTW for AMD EPYC processor based on Naples, Rome and future generation architectures.

8.1.1.1. Download the latest stable release of AMD Optimized FFTW from the link https://github.com/amd/amd-fftw

8.1.1.2. Depending on the target system, and build environment, one would have to enable/disable suitable configure options. Please set PATH and LD_LIBRARY_PATH appropriately to the MPI installation.
The following steps provide instructions for compiling 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 LDFLAGS=”-
no-pie”
```

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 LDFLAGS=”-no-pie”
```

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 LDFLAGS=”-no-pie”
```

Note: Quad Precision not supported in AOCC as of version 2.2

8.1.1.3. `$ make`
8.1.1.4. `$ make install`

6.1.2. Using pre-built binaries

AMD optimized FFTW library binaries for Linux can be found in the following links.

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

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

6.2. Usage

Sample programs demonstrating usage of FFTW APIs and performance benchmarking can be found under the tests/ and mpi/ directories directory of FFTW source.

$ cd fftw-3.3.8/tests   //Sample programs for single-threaded and multi-threaded FFTW

$ cd fftw-3.3.8/mpi   //Sample program for MPI FFTW

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 as well as vector variants of the core math functions. AMD LibM is a C library, which users can link into their applications to replace compiler-provided math functions. Applications can link into AMD LibM library and invoke math functions instead of compiler's math functions for better accuracy and performance.

Latest AMD LibM includes the 'alpha version' of vector variants for the core math functions; power, exponential, logarithmic and trigonometric. Few caveats of the vector variants are listed below.

- Vector variants are relaxed versions of the respective math functions w.r.t accuracy.
- The routines take advantage of the AMD64 architecture for 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 FPU.
- The vector routines will have to be invoked using C intrinsics or from x86 assembly.

Vector variants can be enabled by using AOCC compiler with '`-ffast-math`' flag and it is highly discouraged to call these functions manually. As these functions expect arguments in `__m128`, `__m128d`, `__m256`, `__m256d` types and user has to manually pack-unpack to/from such format.
However, the symbols are enabled in library and the signatures follow the naming convention.

`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`', '`expf`' etc.

For example, single precision 4 element version of exp has signature
`__m128 vrs4_expf(__m128 x)`

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

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 standard C99 function call and naming convention, just needed to be linked with `amdlibm` before standard '`libm`.

Example:

```
$ export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/path/to/amdlibm
$ clang -Wall -std=c99 myprogram.c -o myprogram -lamdlibm -lm
```

OR

```
$ gcc -Wall -std=c99 myprogram.c -o myprogram -lamdlibm -lm
```

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 link.

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

Also, LibM binary can be installed from the 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, Random Number Generator and AMD Secure RNG

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

Note: In this release, AMD LibM binary compiled with GCC is available in the above links. AOCC compiled LibM will be available in a future release.

7.2. Usage

In order to use AMD LibM in your application, follow the below steps.

- Include 'math.h' like standard way to use the C Standard library math functions
- Link in the appropriate version of the library in your program

The Linux libraries might sometimes have a dependency on system math library. When linking AMD LibM, ensure it precedes system math library in the link order i.e., "-lamdlibm" should come before "-lm". Explicit linking of system math library is required when using GCC/AOCC compiler. With g++ compiler (for C++), this is not needed.

```
$ export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/path/to/amdlibm
$ clang -Wall -std=c99 myprogram.c -o myprogram -lamdlibm -lm
$ gcc -Wall -std=c99 myprogram.c -o myprogram -lamdlibm -lm
```

To call vector calls, one has to depend on compiler flag '-ffastmath.
However, though not recommended, one can call the functions directly with manual packing and unpacking. In order to invoke the vector functions directly, one must include the header file 'amdlibm_vec.h'. The following program shows such an example with both returning as well as storing the values in an array. For simplicity the size and other checks are omitted from example.

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/amdlibm
$ clang -Wall -std=c99 -ffastmath myprogram.c -o myprogram -lamdlibm -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 AMD Zen architecture CPUs. This source code of this variant of memcpy is being released and can be compiled and linked against the application which make use of memcpy of glibc.

8.1. Building AMD optimized memcpy

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

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

1. After installing AOCL master installer, check for `amd-memcpy` in the root directory
2. 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 making use of “memcpy” of glibc can link the libamd_memcpy.so library with the below 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 parallel distributed memory machines. It depends on external libraries including BLAS and LAPACK for Linear Algebra computations. AMD's optimized version of ScaLAPACK enables using BLIS and libFLAME library that have optimized dense matrix functions and solvers for AMD EPYC^{TM} processor family CPUs.

9.1. Installation

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

9.1.1. Build ScaLAPACK from source

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

Prerequisites: Building AMD optimized ScaLAPCK library requires linking to following Libraries installed either 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. Static Library

   With GCC(default)

   a. Edit 'SLMake.inc' which contains the build configuration. Update paths of AMD optimized BLIS and libFLAME libraries.

```
BLASLIB_PATH   := <Set the Directory Path where BLIS is installed>
LAPACKLIB_PATH := <Set the Directory Path where libFLAME is installed>

BLASLIB      = $(BLASLIB_PATH)/libblis.a
LAPACKLIB    = $(LAPACKLIB_PATH)/libflame.a
```

   b. Ensure following flags are enabled for GCC in SLMake.inc

```
FCFLAGS      = -cpp -DUSE_BLAS -DF2C -O3
```

   c. Set PATH and LD_LIBRARY_PATH appropriately to the MPI installation.

   d. `$ make clean`

   e. `$ make`

   With AOCC

   a. Edit 'SLMake.inc' which contains the build configuration. Update paths of AMD optimized BLIS and libFLAME libraries.

```
BLASLIB_PATH   := <Set the Directory Path where BLIS is installed>
LAPACKLIB_PATH := <Set the Directory Path where libFLAME is installed>

BLASLIB      = $(BLASLIB_PATH)/libblis.a
LAPACKLIB    = $(LAPACKLIB_PATH)/libflame.a
```

b. Edit SLMake.inc to enable following preprocessors for AOCC build of ScaLAPACK

```
FCFLAGS      = -cpp -DF2C_COMPLEX -DF2C -O3
```

c. Set PATH and LD_LIBRARY_PATH appropriately to the MPI installation.

f. `$ make clean`

g. `$ make`

Note: Another option to set the FCFLAGS as mentioned in (b) is to use the same while running make

`$ make FCFLAGS="-cpp -DF2C_COMPLEX -DF2C -O3"`

4. Shared library using CMake
    a. Create a new directory, say build
       `$ mkdir build`
    b. `$ cd build`
    c. **With GCC**

```
$ 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
```

With AOCC

```
$ 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 \
   -DUSE_DOTC_WRAPPER=ON
```

    d. `$ make -j`
    e. On successful build, Shared library libscalapack.so will be copied in lib\libscalapack.so

9.1.2. Using pre-built binaries

AMD optimized ScaLAPACK library binaries for Linux can be found in the following links.

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, Random Number Generator and AMD Secure RNG

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

9.2. Usage

Applications demonstrating usage of ScaLAPACK APIs can be found under the TESTING directory of ScaLAPACK source package.

```
$ cd scalapack/TESTING
```

10. AMD Random Number Generator

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, users can supply a custom-built generator as the base generator for all the distribution generators.

10.1. Installation

AMD Random Number Generator binary for Linux can be found in the following link.

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

Also, the Random Number Generator 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 AMD Random Number Generator library in your application, you just need to link the library while building the application

Following is a sample Makefile for an application that uses 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 Random Number Generator (RNG) is a library that provides APIs to access the cryptographically secure random numbers generated by AMD's hardware-based random number generator implementation. These are highly quality robust random numbers designed to be suitable for 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 either 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 link.

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 Random Number Generator library.

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

11.2. Usage

Following are the source files included in the AMD Secure RNG package

1. include/secrng.h : Header file that has declaration of all the library APIs.
2. src_lib/secrng.c : Has the implementation of the APIs
3. src_test/secrng_test.c : Test application to test all the library APIs
4. 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.

Below code snippet shows 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 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 operations between a matrix in sparse format and a vector in dense format.
- Sparse Auxiliary Functions describe available helper functions that are required for subsequent library calls.

12.1. Storage Formats

aocl-sparse supports following storage formats for sparse matrices:

12.1.1. CSR storage format

The Compressed Sparse Row (CSR) storage format represents a m*n matrix by

m	number of rows (integer).
n	number of columns (integer).
nnz	number of non-zero elements (integer).
csr_val	array of nnz elements containing the data (floating point).
csr_row_ptr	array of m+1 elements that point to the start of every row (integer).
csr_col_ind	array of nnz elements containing the column indices (integer).

The CSR matrix is expected to be sorted by column indices within each row. Furthermore, each pair of indices should appear only once. Consider the following 3×5 matrix and the corresponding CSR structures, with m=3, n=5 and nnz=8 using zero based indexing:

$$A = \begin{pmatrix} 1.0 & 2.0 & 0.0 & 3.0 & 0.0 \\ 0.0 & 4.0 & 5.0 & 0.0 & 0.0 \\ 6.0 & 0.0 & 0.0 & 7.0 & 8.0 \end{pmatrix}$$

csr_val[8] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0}

csr_row_ptr[4] = {0, 3, 5, 8}

csr_col_ind[8] = {0, 1, 3, 1, 2, 0, 3, 4}

12.1.2. ELLPACK storage format

The Ellpack-Itpack (ELL) storage format represents a m×n matrix by

m	number of rows (integer).
n	number of columns (integer).
ell_width	maximum number of non-zero elements per row (integer)
ell_val	array of m times ell_width elements containing the data (floating point).
ell_col_ind	array of m times ell_width elements containing the column indices (integer).

The ELL matrix is assumed to be stored in row-major format. Rows with less than ell_width non-zero elements are padded with zeros (ell_val) and −1 (ell_col_ind). Consider the following 3×5 matrix and the corresponding ELL structures, with m=3,n=5 and ell_width=3 using zero based indexing:

$$A = \begin{pmatrix} 1.0 & 2.0 & 0.0 & 3.0 & 0.0 \\ 0.0 & 4.0 & 5.0 & 0.0 & 0.0 \\ 6.0 & 0.0 & 0.0 & 7.0 & 8.0 \end{pmatrix}$$

ell_val[9] = {1.0, 2.0, 3.0, 4.0, 5.0, 0.0, 6.0, 7.0, 8.0}

ell_col_ind[9] = {0, 1, 3, 1, 2, -1, 0, 3, 4}

12.2. Types

Type name	Description
aoclsparse_mat_descr	Structure holding all properties of a matrix. It must be initialized using aoclsparse_create_mat_descr() and the returned descriptor must be passed to all subsequent library calls that involve the matrix. It should be destroyed at the end using aoclsparse_destroy_mat_descr()
aoclsparse_index_base	Specify the index base of the indices. For a given aoclsparse_mat_descr, the aoclsparse_index_base can be set using aoclsparse_set_mat_index_base(). The current aoclsparse_index_base of a matrix can be obtained by aoclsparse_get_mat_index_base(). **Values**:

	`Enumerator aoclsparse_index_base_zero = 0 zero based indexing (default)`
	
	`Enumerator aoclsparse_index_base_one = 1 one based indexing.`
	
	The current release supports only zero-based indexing

aoclsparse_matrix_type	Specify the type of a matrix. For a given aoclsparse_mat_descr, the aoclsparse_matrix_type can be set using aoclsparse_set_mat_type(). The current aoclsparse_matrix_type of a matrix can be obtained by aoclsparse_get_mat_type().
	
	Values:
	
	`enumerator aoclsparse_matrix_type_general = 0 (default)`
	
	`enumerator aoclsparse_matrix_type_symmetric = 1`
	
	`enumerator aoclsparse_matrix_type_hermitian = 2`
	
	`enumerator aoclsparse_matrix_type_triangular = 3`
	
	The current release supports only general matrix types

aoclsparse_operation	Specify whether the matrix is to be transposed or not.
	
	Values:
	
	`enumerator aoclsparse_operation_none = 111 ->Operate with matrix (default)`
	
	`enumerator aoclsparse_operation_transpose = 112 -> Operate with transpose.`
	
	`enumerator aoclsparse_operation_conjugate_transpose = 113 -> Operate with conj. transpose.`
	
	The current release supports only aoclsparse_operation_none

aoclsparse_status	List of aoclsparse status codes definition
	
	Values:
	
	`enumerator aoclsparse_status_success = 0`
	
	`enumerator aoclsparse_status_not_implemented = 1`
	
	`enumerator aoclsparse_status_invalid_pointer = 2`

	`enumerator aoclsparse_status_invalid_size = 3`
	`enumerator aoclsparse_status_internal_error  = 4,`
	`enumerator aoclsparse_status_invalid_value = 5`

12.3. Sparse functions

This release of aocl-sparse supports following Sparse functions

12.3.1. Sparse Level 2 Functions

Function name	**Description**
aoclsparse_Xcsrmv()	Performs following operation: $y := \alpha \cdot op(A) \cdot x + \beta \cdot y$ where A is a a sparse m×n matrix, defined in CSR storage format, x dense vector, y result dense vector. α and β are scalars Datatypes supported: Single and double precision.
aoclsparse_Xellmv()	Performs following operation: $y := \alpha \cdot op(A) \cdot x + \beta \cdot y$ where A is a a sparse m×n matrix, defined in ELLPACK storage format, x dense vector, y result dense vector. α and β are scalars Datatypes supported: Single and double precision.

Notes on current AOCL-Sparse release:

1. This release supports only general matrices. Symmetric and triangular matrices should be converted into general matrices before invoking these API's
2. For CSRMV and ELLMV ($y := \alpha \cdot op(A) \cdot x + \beta \cdot y$), op(A) = Transpose/Conjugate of (A) is not supported in the current release.
3. This release supports only zero-based indexing of matrices in CSR and ELLPACK formats.

12.3.2. Auxiliary Functions

Function name	Description
aoclsparse_get_version()	Gets the aoclSPARSE library version number.
aoclsparse_create_mat_descr()	Creates a matrix descriptor. It initializes aoclsparse_matrix_type to aoclsparse_matrix_type_general and aoclsparse_index_base to aoclsparse_index_base_zero.

aoclsparse_destroy_mat_descr()	Destroys a matrix descriptor and releases all resources used by the descriptor.
aoclsparse_set_mat_index_base()	Sets the index base of a matrix descriptor. Valid options are aoclsparse_index_base_zero or aoclsparse_index_base_one.
aoclsparse_get_mat_index_base()	Returns the index base of a matrix descriptor.
aoclsparse_set_mat_type()	Sets the matrix type of a matrix descriptor. Valid matrix types are aoclsparse_matrix_type_general, aoclsparse_matrix_type_symmetric, aoclsparse_matrix_type_hermitian or aoclsparse_matrix_type_triangular.
aoclsparse_get_mat_type()	Returns the matrix type of a matrix descriptor.

12.4. Installation

12.4.1. Build aocl-sparse from source

The following instructions can be used to build aocl-sparse from source. Furthermore, the following compile-time dependencies must be met

- git
- CMake 3.5 or later
- libboost-program-options (optional, for running tests)

Download aocl-sparse

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

```
$ git clone https://github.com/amd/aocl-sparse.git
$ cd aocl-sparse
```

Build aocl-sparse

Below are steps to build different packages of the library, including dependencies and clients. aocl-sparse can be built using the following commands:

```
# Create and change to build directory
$ mkdir -p build/release ; cd build/release
```

```
# With GCC(Default)
# Default install path is /opt/aoclsparse/, use -DCMAKE_INSTALL_PREFIX=<path> to choose custom path
$ cmake ../..

# With AOCC
# Default install path is /opt/aoclsparse/, use -DCMAKE_INSTALL_PREFIX=<path to install>
$ cmake ../.. -DCMAKE_CXX_COMPILER=clang++

# Compile aocl-sparse library
$ make -j$(nproc)

# Install aocl-sparse to /opt/aoclsparse/
$ make install
```

Boost is required in order to build aocl-sparse test application. aocl-sparse with dependencies and clients can be built using the following commands:

```
# Install boost on e.g. Ubuntu
$ apt install libboost-program-options-dev

# Change to build directory
$ cd build/release

# With GCC(Default)
# Default install path is /opt/aoclsparse, use -DCMAKE_INSTALL_PREFIX=<path> to choose custom path
$ cmake ../.. -DBUILD_CLIENTS_BENCHMARKS=ON

# With AOCC
# Default install path is /opt/aoclsparse/, use -DCMAKE_INSTALL_PREFIX=<path> to adjust it
$ cmake ../.. -DCMAKE_CXX_COMPILER=clang++ \
            -DBUILD_CLIENTS_BENCHMARKS=ON

# Compile aocl-sparse library
$ make -j$(nproc)

# Install aocl-sparse to /opt/aoclsparse
$ make install
```

12.4.2. Simple Test

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

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

# Execute aocl-sparse example by running CSR-SPMV on randomly generated matrix
$ ./aoclsparse_bench -f csrmv --precision d -m 1000 -n 1000 -z 4000 -v 1
```

12.4.3. Using prebuilt libraries

AMD optimized aocl-sparse library binaries for Linux can be found in the following links.

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

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

12.5. Usage

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

```
$ cd aocl-sparse/test/
```

Use by Applications

To use aocl-sparse in your application, you just need to 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>  -L<path-to aocl-sparse-library>/libaoclsparse.so
-o test_aoclsparse.x
```

Below is a sample cpp file depicting 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;
}
```

Example compilation command, for the above code, with gcc compiler:

```
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, one can build a package with multiple versions, configurations, platforms, and compilers, and all these builds can coexist on the same machine.

Note 1: Starting AOCL 2.2 release, Spack recipes for AMD optimized libraries of BLIS, libFLAME and FFTW will be available in new GitHub repository https://github.com/amd/spack. The earlier AMD Spack Github repo https://github.com/amd/aocl-spack is deprecated.

Note 2: AOCL 2.2 packages are tested using Spack release version - v0.14

13.1. AOCL Spack Environment Setup

Clone AMD Spack GitHub repository

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

Set environment path for Spack shell.

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

13.2. Install AOCL packages

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

Current release of AOCL supports install of AMD optimized libraries of BLIS, libFLAME and FFTW.

13.2.1. Install amdblis Spack package

```
$ spack install amdblis
```

13.2.2. Install amdlibflame Spack package

```
$ spack install amdlibflame
```

13.2.3. Install amdfftw Spack package

```
$ spack install amdfftw
```

13.3. Spack useful commands

Here are few useful Spack commands to get additional information on the spack packages. Basic Spack details on the link.

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
```

Under BLIS installation directory, user will get .spack directory which contains below files or directories:
. 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

To install other versions of amdblis package, use @<version-number>
```
$ spack install -v amdblis@2.1
```

To 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
```

13.4. Uninstall AOCL 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 cpu-intensive 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.

An optimized HPL binary for AMD EPYC CPUs is available under the download section of https://developer.amd.com/amd-aocl/blas-library/.

15. AOCL Tuning Guidelines

This section provides tuning recommendations for AOCL to derive best optimal performance on AMD EPYC^{TM} and future generation architectures.

15.1. BLIS DGEMM multi-thread tuning

AMD Rome

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

Thread Size upto 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 best DGEMM multi-thread performance on 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 block sizes used by BLIS. Enabling and disabling these macros causes choosing the appropriate block sizes that BLIS operates on.

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 multithread option as mention in section Multi-threaded BLIS
3. Link generated BLIS library to your application and execute
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 section Multi-threaded BLIS
3. Link 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 specified block sizes[4].

Many high-performance computing (HPC) and scientific applications and benchmarks run on high end cluster of machines, each with multiple cores. They run programs with multiple instances which could be 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 block sizes specified would 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.

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

AMD Rome

1. Open the file *bli_family_zen2.h* under BLIS source
   $ cd "*config/zen2/ bli_family_zen2.h*"

2. For applications/benchmarks running in multi-instance mode and using multi-threaded BLIS, ensure 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 when using multi-threaded BLIS.
   ```
   #define AOCL_BLIS_MULTIINSTANCE       0
   ```

3. For applications/benchmarks running in multi-instance mode and using single-threaded BLIS, set the macro, AOCL_BLIS_MULTIINSTANCE to 1. Recompile BLIS source and link it to application. HPL benchmark is found to generate better performance numbers using this setting when using single-threaded BLIS.

AMD Naples

1. Open the file *bli_cntx_init_zen.c* under 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 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 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 selective packing for GEMM, when one- or two-dimensions of matrix is exceedingly small. This feature is only available when "sup handling" is enabled (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

Below are the tuning guidelines to get 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 AMD CPU specific improvements with 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 that may benefit performance of transpose operations in case of very large FFT problem sizes. This feature is to be used only when running as 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 would benefit most of the MPI problem types and sizes.
5. For best performance, please use the "-opatient" planner flag of FFTW.
   Example of running FFTW bench test application with "-opatient" planner flag is as below:-
   ```
   $ ./bench -opatient -s icf65536
   ```
   where -s option is for speed/performance run and icf options stand for in-place, complex data-type, forward transform.

16. Appendix

16.1. Check AMD Server Processor Architecture

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

1. Run lscpu command
   ```
   $ lscpu
   ```

2. Check the values "CPU family" and "Model" fields

3. For Naples
   ```
   cpu family    : 23
   model         : Values in the range <1 – 47>
   ```

4. For Rome
   ```
   cpu family    : 23
   model         : Values in the range < 48 – 63>
   ```

17. Technical Support and Forums

For questions and issues about AOCL, one can reach us on the following email-id 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/