

AMD Optimizing CPU Libraries User Guide

Version 2.1

Table of Contents

1. Introduction	4
2. Supported Operating Systems and Compilers	5
3. BLIS library for AMD	6
3.1. Installation	6
3.1.1. Build BLIS from source	6
3.1.1.1. Single-thread BLIS	6
3.1.1.2. Multi-threaded BLIS	6
3.1.2. Using pre-built binaries.....	7
3.2. Usage.....	7
3.2.1. BLIS Usage in FORTRAN.....	8
3.2.2. BLIS Usage in C through BLAS and CBLAS APIs	9
4. libFLAME library for AMD	12
4.1. Installation	12
4.1.1. Build libFLAME from source.....	12
4.1.2. Using pre-built binaries.....	13
4.2. Usage.....	13
5. FFTW library for AMD	14
5.1. Installation	14
5.1.1. Build FFTW from source.....	14
5.1.2. Using pre-built binaries.....	15
5.2. Usage.....	15
6. AMD LibM	16
6.1. Installation	18
6.2. Usage.....	19
7. ScaLAPACK library for AMD.....	20
7.1. Installation	20
7.1.1. Build ScaLAPACK from source	20
7.1.2. Using pre-built binaries.....	20
7.2. Usage.....	21
8. AMD Random Number Generator	22
8.1. Installation	22
8.2. Usage.....	22

9.	AMD Secure RNG	23
9.1.	Installation	23
9.2.	Usage.....	23
10.	AOCL Spack Recipes	25
10.1.	AOCL Spack Environment Setup	25
10.1.1.	Option 1: Install AOCL Spack Enviroment via Automated Script	25
10.1.2.	Option 2: Install AOCL Spack Environment via manual steps	25
10.2.	Install AOCL packages	26
10.2.1.	Install AMD BLIS package	26
10.3.	Uninstall AOCL Packages.....	27
11.	Applications integrated to AOCL.....	28
11.1.	High-Performance LINPACK Benchmark (HPL)	28
12.	AOCL Tuning Guidelines.....	29
12.1.	BLIS DGEMM multi-thread tuning.....	29
12.2.	BLIS DGEMM block size tuning for single and multi-instance mode	30
12.3.	AMD Optimized FFTW Tuning Guidelines.....	32
13.	Appendix	33
13.1.	Check AMD Server Processor Architecture.....	33
14.	Technical Support and Forums	34
15.	References	35

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 seven 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.

In addition, we provide Spack(<https://spack.io/>) based recipes for installing BLIS, libFLAME and FFTW libraries.

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 2.1 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 Compilers

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 2.1 (<https://developer.amd.com/amd-aocc/>)

3. 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.

3.1. Installation

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

3.1.1. Build BLIS from source

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

3.1.1.1. 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. 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.

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

3.1.1.2. 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. 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.

With GCC

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

With AOCC

```
$ ./configure --enable-cblas --disable-sup-handling --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

For more information on multi-threaded implementation in BLIS refer [here](#).

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

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

3.2. 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-to-BLIS-library>/libblis.a -o test_blis.x
```

With Dynamic Library:

```
gcc test_blis.c -I<path-to-BLIS-header> -L<path-to-BLIS-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.

3.2.1. 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
```

3.2.2. 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. 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.

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.

4.1. Installation

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

4.1.1. 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 --prefix=<your-install-dir>
```

With AOCC

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

'Argument list too long' build error

On some build environments, compiling libFLAME might throw '*Argument list too long*' error message when *make* tries to archive the object files to library. In such case, in order to reduce the length of argument list, enable the option "`--enable-max-arg-list-hack`"

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

3. Make and install. By default the library will be installed to \$HOME/flame

- \$ make
- \$ make install

4.1.2. 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/>

4.2. 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>
$ ./test_libflame.x
```

Run libFLAME C++ Template API tests as below

From the libFLAME source directory,

With GCC

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

With AOCC

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

5. 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.

5.1. Installation

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

5.1.1. 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.

- 7.1.1.1. Download the latest stable release of AMD Optimized FFTW from the link

<https://github.com/amd/amd-fftw>

- 7.1.1.2. 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 --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 --prefix=<your-install-dir>
```

Long double FFTW libraries

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

Quad Precision FFTW libraries

```
$ ./configure --enable-static --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 --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 --prefix=<your-install-dir> CC=clang F77=flang LDFLAGS="-no-pie"
```

Long double FFTW libraries

```
$ ./configure --enable-static --enable-shared --enable-openmp --enable-mpi --enable-long-double --enable-amd-opt --prefix=<your-install-dir> CC=clang F77=flang LDFLAGS="-no-pie"
```

Note: Quad Precision not supported in AOCC as of version 2.1

7.1.1.3. \$ make

7.1.1.4. \$ make install

5.1.2. 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, Random Number Generator and AMD Secure RNG

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

5.2. Usage

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

```
$ cd fftw-3.3.8/tests
```

6. 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_exp2f, vrs4_exp2f, vrs4_exp10f, vrs4_expm1f
* vrsa_exp2f, 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, sinh, 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
```

6.1. 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.

6.2. 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(ip[3], ip[2], ip[1], ip[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
```

7. 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™ processor family CPUs.

7.1. Installation

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

7.1.1. Build ScaLAPACK from source

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

Prerequisites: Building AMD optimized ScaLAPACK library requires linking to following Libraries installed either using pre-built binaries or built from source:

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

1. git clone <https://github.com/amd/scalapack.git>
2. \$ cd scalapack
3. 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
```

4. Set PATH and LD_LIBRARY_PATH appropriately to the MPI installation.
5. \$ make clean
6. \$ make

7.1.2. Using pre-built binaries

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, Random Number Generator and AMD Secure RNG

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

7.2. Usage

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

```
$ cd scalapack/TESTING
```

8. 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.

8.1. 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 and AMD Secure RNG

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

8.2. 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.

9. 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.

9.1. 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 and AMD Random Number Generator library.

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

9.2. 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");
}
```

10. 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. - <https://spack.io/>

Basic steps for installing AOCL Spack are in following sub-sections. For detailed steps, please refer README file at following link.

<https://github.com/amd/aocl-spack/AOCL-README>

Note: AOCL packages are tested using Spack release version - v0.12

10.1. AOCL Spack Environment Setup

AOCL Spack environment setup can be done through an automated script (spack_set_env.sh) or via manual setup steps

10.1.1. Option 1: Install AOCL Spack Enviroment via Automated Script

1. Download AOCL Spack tar package from URL <https://github.com/amd/aocl-spack/releases>

2. Extract aocl_spack_recipe.tar package

```
$ tar -xf aocl_spack_recipe.tar
```

3. Script usage:

3.1. If Spack tool not present in machine:

Run the spack environment script

```
$ bash spack_set_env.sh -t spack_recipes.tar
```

Add spack to system PATH

```
$ source <Current dir>/SPACK_SRC/spack/share/spack/setup-env.sh
```

3.2. If Spack tool already present in machine:

Run the spack environment script

```
$ bash spack_set_env.sh -t spack_recipes.tar -s <Spack install path>
```

10.1.2. Option 2: Install AOCL Spack Environment via manual steps

For manual steps of setting up the Spack environment, please refer README file at following link.

<https://github.com/amd/aocl-spack/AOCL-README>

10.2. Install AOCL packages

AOCL package, which consists of amd.Blis, amd.libFlame and amd.fftw can be installed using below procedure as shown for BLIS package:

10.2.1. Install AMD BLIS package

Display blis package info and supported versions

```
$ spack info amd.blis
```

Install amd blis latest version (For AOCL 2.1 release, this will be BLIS for AMD version 2.1)

```
$ spack install amd.blis
```

Verify contents installed contents

```
$ spack spec amd.blis
```

Go to BLIS install-directory

```
$ spack cd -i amd.blis
```

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

- . build.env - captures build environment details
- . build.out - 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 amd blis package use @ (supported versions of BLIS are 2.1, 2.0, 1.3, 1.2 and 1.0):

```
$ spack install -v amd.blis@1.3
```

Build and install Blis 1.2 with openmp multithreading using AOCC 2.1 compiler:

```
$ spack install amd.blis@1.2 threads=openmp %clang@9.0.0
```

Build and install Blis 1.0 with pthread multithreading using GCC compiler:

```
$ spack install amd.blis@1.0 threads=threads %gcc@9.2.0
```

Note: AMD FFTW package is delivered for single and double precision. Default installation creates double precision binaries.

For single precision, use below command:

```
$ spack install amd.fftw +single
```

10.3. Uninstall AOCL Packages

The following commands show how AMD BLIS package can be uninstalled. Similar procedure can be used for other libraries, libFLAME and FFTW.

Uninstall BLIS default package

```
$ spack uninstall amd.blis
```

Uninstall BLIS based out of different versions:

```
$ spack uninstall amd.blis@2.0
```

Uninstall BLIS based out of hash values:

```
$ spack uninstall amd.blis/43reafx
```

Uninstall BLIS based out of different namespace:

```
$ spack uninstall builtin.blis@0.6.0
```

11. 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.

11.1. 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/>. This has been validated on Ubuntu 19.04.

12. AOCL Tuning Guidelines

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

12.1. 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
```

12.2. 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 );
```

12.3. 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. 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.

13. Appendix

13.1. 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>

14. Technical Support and Forums

For questions and issues about AOCL, one can reach us on the following email-id
toolchainsupport@amd.com

15. 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/>

DISCLAIMER

The information contained herein is for informational purposes only, and is subject to change without notice. While every precaution has been taken in the preparation of this document, it may contain technical inaccuracies, omissions and typographical errors, and AMD is under no obligation to update or otherwise correct this information. Advanced Micro Devices, Inc. makes no representations or warranties with respect to the accuracy or completeness of the contents of this document, and assumes no liability of any kind, including the implied warranties of noninfringement, merchantability or fitness for particular purposes, with respect to the operation or use of AMD hardware, software or other products described herein. No license, including implied or arising by estoppel, to any intellectual property rights is granted by this document. Terms and limitations applicable to the purchase or use of AMD's products are as set forth in a signed agreement between the parties or in AMD's Standard Terms and Conditions of Sale.

AMD, the AMD Arrow logo, EPYC and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

© 2018-20 Advanced Micro Devices, Inc. All rights reserved.