AOCL User Guide
**Trademarks**

AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc.

Dolby is a trademark of Dolby Laboratories.

ENERGY STAR is a registered trademark of the U.S. Environmental Protection Agency.

HDMI is a trademark of HDMI Licensing, LLC.

HyperTransport is a licensed trademark of the HyperTransport Technology Consortium.

Microsoft, Windows, Windows Vista, Windows Server, Visual Studio, and DirectX are registered trademarks of Microsoft Corporation.

MMX is a trademark of Intel Corporation.

OpenCL is a trademark of Apple Inc. used by permission by Khronos.

PCIe is a registered trademark of PCI-Special Interest Group (PCI-SIG).

Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

**Dolby Laboratories, Inc.**

Manufactured under license from Dolby Laboratories.

**Rovi Corporation**

This device is protected by U.S. patents and other intellectual property rights. The use of Rovi Corporation’s copy protection technology in the device must be authorized by Rovi Corporation and is intended for home and other limited pay-per-view uses only, unless otherwise authorized in writing by Rovi Corporation.

Reverse engineering or disassembly is prohibited.

USE OF THIS PRODUCT IN ANY MANNER THAT COMPLIES WITH THE MPEG-2 STANDARD IS EXPRESSLY PROHIBITED WITHOUT A LICENSE UNDER APPLICABLE PATENTS IN THE MPEG-2 PATENT PORTFOLIO, WHICH LICENSE IS AVAILABLE FROM MPEG LA, L.L.C., 6312 S. FIDDLERS GREEN CIRCLE, SUITE 400E, GREENWOOD VILLAGE, COLORADO 80111.
## Contents

**Revision History** ........................................................................................................... 9

**Chapter 1** Introduction .................................................................................................... 10

**Chapter 2** Supported OS and Compilers ........................................................................... 11
  
  2.1 Operating Systems ........................................................................................................ 11
  
  2.2 Compilers .................................................................................................................... 11
  
  2.3 Library ........................................................................................................................ 11
  
  2.4 Message Passing Interface (MPI) .................................................................................. 11
  
  2.5 Programming Language .............................................................................................. 11

**Chapter 3** Installing AOCL ............................................................................................... 12
  
  3.1 Building from Source .................................................................................................... 12
  
  3.2 Installing AOCL Binary Packages ............................................................................... 12
    
    3.2.1 Using Master Package ............................................................................................ 12
    
    3.2.2 Using Library Package ............................................................................................ 13
    
    3.2.3 Using Debian and RPM Packages ......................................................................... 13
    
    3.2.4 Using Windows Packages ....................................................................................... 15

**Chapter 4** BLIS Library for AMD ...................................................................................... 17
  
  4.1 Installation on Linux ....................................................................................................... 17
    
    4.1.1 Build BLIS from Source ......................................................................................... 17
    
    4.1.2 Using Pre-built Binaries ......................................................................................... 19
  
  4.2 Application Development Using BLIS ........................................................................... 19
    
    4.2.1 API Compatibility Layers (Calling BLIS) ............................................................... 19
    
    4.2.2 API Compatibility - Advance Options .................................................................... 21
    
    4.2.3 Linking Application with BLIS ............................................................................... 21
    
    4.2.4 Example Application - BLIS Usage in FORTRAN ............................................... 22
    
    4.2.5 BLIS Usage in C .................................................................................................... 24
  
  4.3 Migrating/Porting .......................................................................................................... 28
  
  4.4 Using BLIS Library Features ....................................................................................... 29
    
    4.4.1 Dynamic Dispatch .................................................................................................. 29
    
    4.4.2 BLIS - Running in-built Test Suite ..................................................................... 30
4.4.3 Testing/Benchmarking ................................................................. 31
4.4.4 BLIS APIs .................................................................................. 33
4.5 Debugging and Troubleshooting ......................................................... 33
4.5.1 Debugging Build Using GDB .......................................................... 33
4.5.2 Viewing Logs .............................................................................. 34
4.5.3 Usages and Limitations ................................................................. 38
4.6 Build BLIS from Source on Windows ................................................ 38
4.6.1 Building BLIS using GUI .............................................................. 39
4.6.2 Building BLIS using Command-line Arguments ................................. 43
4.6.3 Building and Running Test Suite .................................................... 43

Chapter 5 libFLAME Library for AMD ................................................. 45
5.1 Installing on Linux .......................................................................... 45
5.1.1 Building libFLAME from Source ................................................. 45
5.1.2 Using Pre-built Libraries .............................................................. 46
5.2 Usage .............................................................................................. 47
5.2.1 Source Directory ....................................................................... 47
5.2.2 Use by Applications ................................................................. 47
5.3 Building libFLAME from Source on Windows ................................. 47
5.3.1 Building libFLAME Using GUI .................................................. 48
5.3.2 Building libFLAME using Command-line Arguments .................... 50
5.3.3 Building and Running Test Suite ............................................... 51

Chapter 6 FFTW Library for AMD ....................................................... 52
6.1 Installing ......................................................................................... 52
6.1.1 Building FFTW from Source on Linux ......................................... 52
6.1.2 Building FFTW from Source on Windows .................................... 54
6.1.3 Using Pre-built Libraries ............................................................. 57
6.2 Usage ............................................................................................ 57
6.2.1 Sample Programs for Single-threaded and Multi-threaded FFTW .... 57
6.2.2 Sample Programs for MPI FFTW ............................................... 58
6.2.3 Additional Options ................................................................. 58

Chapter 7 AMD LibM ............................................................................ 60
7.1 Installation on Linux ......................................................... 63
7.2 Compiling AMD LibM ......................................................... 63
7.3 Usage ................................................................. 64
7.4 Building AMD LibM on Windows ........................................ 65

Chapter 8 ScaLAPACK Library for AMD ........................................ 67
  8.1 Installation ............................................................ 67
     8.1.1 Building ScaLAPACK from Source ................................ 67
     8.1.2 Using Pre-built Libraries ........................................... 70
  8.2 Usage ................................................................. 70
  8.3 Building ScaLAPACK from Source on Windows ....................... 70
     8.3.1 Building ScaLAPACK Using GUI .................................... 71
     8.3.2 Building ScaLAPACK using Command-line Arguments ............ 73

Chapter 9 AMD Random Number Generator ..................................... 74
  9.1 Installation ............................................................ 74
  9.2 Usage ................................................................. 74

Chapter 10 AMD Secure RNG .................................................... 75
  10.1 Installation ............................................................ 75
  10.2 Usage ................................................................. 75

Chapter 11 AOCL-Sparse .......................................................... 77
  11.1 Installation ............................................................ 78
     11.1.1 Building AOCL-Sparse from Source ............................... 78
     11.1.2 Simple Test ......................................................... 79
     11.1.3 Using Pre-built Libraries ........................................... 80
  11.2 Usage ................................................................. 80
     11.2.1 Use by Applications ................................................ 80
  11.3 Build AOCL-Sparse from Source on Windows ......................... 82
     11.3.1 Building AOCL-Sparse Using GUI ............................... 82
     11.3.2 Building AOCL-Sparse using Command-line Arguments .......... 83

Chapter 12 AOCL Spack Recipes .................................................. 85
  12.1 Setting Up AOCL Spack Environment ................................ 85
  12.2 Installing AOCL Packages ............................................. 85
12.2.1 Installing amdblis Spack Package ........................................... 85
12.2.2 Installing amdlibflame Spack Package ................................... 85
12.2.3 Installing amdfftw Spack Package ........................................ 86
12.2.4 Installing amdscalapack Spack Package .................................. 86
12.2.5 Installing amdsparse Spack Package ....................................... 86
12.2.6 Installing amdlibm Spack Package ......................................... 86
12.2.7 Installing Legacy AOCL versions ......................................... 86
12.3 Spack Commands ................................................................. 86
12.4 Uninstalling AOCL Packages ................................................ 87

Chapter 13 Integration with AOCL .................................................. 88
13.1 High-performance LINPACK Benchmark (HPL) ............................ 88
13.1.1 Configuring HPL.dat ......................................................... 88
13.1.2 Running the Benchmark ................................................... 88
13.2 MUMPS Sparse Solver Library .............................................. 89
13.2.1 Enabling AOCL with MUMPS .......................................... 90

Chapter 14 AOCL Tuning Guidelines ............................................... 95
14.1 AOCL Dynamic ................................................................. 95
14.1.1 Limitations ................................................................. 95
14.2 BLIS DGEMM Multi-thread Tuning ......................................... 95
14.2.1 Library Usage Scenarios .................................................. 96
14.2.2 Architecture Specific Tuning ............................................. 97
14.3 BLIS DGEMM Block-size Tuning ............................................ 98
14.4 Performance Suggestions for Skinny Matrices ........................... 100
14.5 AMD Optimized FFTW Tuning Guidelines .................................. 100

Chapter 15 Support ................................................................. 102

Chapter 16 References ............................................................. 103

Appendix .................................................................................. 104

Check AMD Server Processor Architecture .................................. 104
Application Notes ................................................................. 104
AMD-FFTW ................................................................. 104
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1.</td>
<td>BLIS API Compatibility Layers</td>
<td>20</td>
</tr>
<tr>
<td>Table 2.</td>
<td>BLIS API Compatibility - Advance Options</td>
<td>21</td>
</tr>
<tr>
<td>Table 3.</td>
<td>BLIS Application - Link Options</td>
<td>21</td>
</tr>
<tr>
<td>Table 4.</td>
<td>Porting to BLIS</td>
<td>29</td>
</tr>
<tr>
<td>Table 5.</td>
<td>BLIS APIs</td>
<td>33</td>
</tr>
<tr>
<td>Table 6.</td>
<td>CMake Config Options</td>
<td>41</td>
</tr>
<tr>
<td>Table 7.</td>
<td>libFLAME Config Options</td>
<td>48</td>
</tr>
<tr>
<td>Table 8.</td>
<td>FFTW Config Options</td>
<td>55</td>
</tr>
<tr>
<td>Table 9.</td>
<td>Compiler and Type of Library</td>
<td>68</td>
</tr>
<tr>
<td>Table 10.</td>
<td>ScALAPACK CMake Parameter List</td>
<td>71</td>
</tr>
<tr>
<td>Table 11.</td>
<td>Compiler and Library Type</td>
<td>79</td>
</tr>
<tr>
<td>Table 12.</td>
<td>AOCL-Sparse CMake Parameter List</td>
<td>82</td>
</tr>
<tr>
<td>Table 13.</td>
<td>Spack Commands</td>
<td>86</td>
</tr>
<tr>
<td>Table 14.</td>
<td>Spack Directory Files</td>
<td>87</td>
</tr>
<tr>
<td>Table 15.</td>
<td>Sample Commands</td>
<td>87</td>
</tr>
<tr>
<td>Table 16.</td>
<td>AOCL Dynamic</td>
<td>95</td>
</tr>
</tbody>
</table>
List of Figures

Figure 1. Sample Run of Function Call Tracing ......................................................... 36
Figure 2. Sample Run with Debug Logs Enabled ....................................................... 37
Figure 3. Debug Logs Showing Input Values of GEMM ............................................. 38
Figure 4. Microsoft Visual Studio Prerequisites ....................................................... 39
Figure 5. CMake Source and Build Folders .............................................................. 40
Figure 6. Set Generator and Compiler ................................................................. 40
Figure 7. CMake Configure and Generate Project Settings ....................................... 42
Figure 8. libFLAME CMake Configurations ............................................................ 50
Figure 9. FFTW CMake Config Options ................................................................. 56
Figure 10. ScaLAPACK CMake Options ................................................................. 72
Figure 11. ScaLAPACK CMake Config Options ...................................................... 72
Figure 12. AOCL-Sparse CMake Config Options ..................................................... 83
# Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Revision</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>December 2021</td>
<td>3.1</td>
<td>Initial version.</td>
</tr>
</tbody>
</table>
Chapter 1 Introduction

AMD Optimizing CPU Libraries (AOCL) are a set of numerical libraries optimized for AMD “Zen”-based processors, including EPYC™, Ryzen™ Threadripper™ PRO, and Ryzen™. This document provides instructions on installing and using all the AMD optimized libraries.

AOCL is comprised of the following eight libraries:

- **BLIS (BLAS Library)** is a portable open-source software framework for performing high-performance Basic Linear Algebra Subprograms (BLAS) functionality.

- **libFLAME (LAPACK)** is a portable library for dense matrix computations that provides the functionality present in the Linear Algebra Package (LAPACK).

- **AMD-FFTW (Fastest Fourier Transform in the West)** is a comprehensive collection of fast C routines for computing the Discrete Fourier Transform (DFT) and various special cases.

- **LibM (AMD Core Math Library)** is a software library containing a collection of basic math functions optimized for x86-64 processor based machines.

- **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 Random Number Generator (RNG)** is a pseudo random number generator library.

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

- **AOCL-Sparse** is a library containing the basic linear algebra subroutines for sparse matrices and vectors optimized for AMD “Zen”-based processors, including EPYC™, Ryzen™ Threadripper™ PRO, and Ryzen™.

Additionally, AMD provides the Spack ([https://spack.io/](https://spack.io/)) recipes for optimally installing BLIS, libFLAME, ScaLAPACK, LibM, FFTW, and Sparse libraries.

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

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

AOCL 3.1 includes several performance improvements for the 3rd Gen AMD EPYC™ microprocessor architecture in addition to prior AMD architectures. Refer Appendix Check AMD Server Processor Architecture to determine the underlying architecture of your AMD system.
Chapter 2  Supported OS and Compilers

This release of AOCL has been validated on the following:

Note: For the supported compiler versions and prerequisites of a specific library, refer to the corresponding sections.

2.1  Operating Systems

• Ubuntu® 20.04 LTS and 21.04
• CentOS 8
• Red Hat® Enterprise Linux® (RHEL) 8.3.1
• SUSE Linux Enterprise Server (SLES) 15 SP3
• Windows Server 2019
• Windows® 10

2.2  Compilers

• GCC 9.2.1, 9.2, 9.3, and 11.1
• AOCC 3.1 and 3.2
• LLVM™ 12 and 13

2.3  Library

Glibc 2.28 and 2.31

2.4  Message Passing Interface (MPI)

OpenMPI 4.1.1

2.5  Programming Language

Python 2.7 and later
Chapter 3 Installing AOCL

3.1 Building from Source

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

- BLIS ([https://github.com/amd/blis](https://github.com/amd/blis))
- libFLAME ([https://github.com/amd/libflame](https://github.com/amd/libflame))
- FFTW ([https://github.com/amd/amd-fftw](https://github.com/amd/amd-fftw))
- LibM ([https://github.com/amd/aocl-libm-ose](https://github.com/amd/aocl-libm-ose))
- ScaLAPACK ([https://github.com/amd/aocl-scalapack](https://github.com/amd/aocl-scalapack))
- AOCL-Sparse ([https://github.com/amd/aocl-sparse](https://github.com/amd/aocl-sparse))

The details on installing from source for each library (including Spack-based install of AOCL libraries) is explained in the later sections.

3.2 Installing AOCL Binary Packages

The section describes the procedure to install AOCL binaries on Linux and Windows.

3.2.1 Using Master Package

Complete the following steps to install the AOCL library suite:

2. Use the command `tar -xvf <aocl-linux-<compiler>-3.1.0.tar.gz>` to untar the package. Locate the installer file `install.sh` in the package.
3. Run `./install.sh` to install the AOCL package (all libraries) to the default INSTALL_PATH: `/home/<username>/amd/aocl/3.1.0`.

Suffix `-h` to `install.sh` to print the usage of the script with all the following supported options:

- `-h` — Print the help.
- `-t` — Custom target directory to install libraries.
- `-l` — Library to be installed.
- `-i` — Select LP64/ILP64 libraries to be set as default.

4. To install the AOCL package in a custom location, use the installer with the option: `-t <CUSTOM_PATH>`. For example, `./install.sh -t /home/<username>`.
5. You can use the master installer to install the individual library out of the master package. The library names used are blis, libflame, libm, scalapack, rng, secrng, and sparse. You can do one of the following:
   • To install a specific library, use the option: `-l <Library name>`. For example, `./install.sh -l blis`.
   • Install the individual library in a path of your choice. For example, `./install.sh -t /home/amd -l libm`.

6. AOCL 3.1 supports AOCL libraries with the following two integer types:
   • LP64 libraries and header files are installed in 
     `/INSTALL_PATH/lib_LP64` and `/INSTALL_PATH/include_LP64` respectively.
   • ILP64 libraries and header files are installed in 
     `/INSTALL_PATH/lib_ILP64` and `/INSTALL_PATH/include_ILP64` respectively.

By default, LP64 libraries and header files are available in `/INSTALL_PATH/lib` and `/INSTALL_PATH/include` respectively.

Suffix `.install.sh` with `-i <lp64/ilp64>` to:
   • Set the LP64 libraries as the default libraries, use the installer with the option: `-i lp64`. For example, `./install.sh -t /home/amd -l blis -i lp64`. This installs only Blis library in the path `/home/amd` and sets LP64 BLIS libraries as the default.
   • Set ILP64 libraries as the default use the installer with the option: `-i ilp64`. For example, `./install.sh -i ilp64`. This installs all AOCL libraries in the default path and sets ILP64 libraries as the default.

### 3.2.2 Using Library Package

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

### 3.2.3 Using Debian and RPM Packages

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

The package name used in the following installation procedure is based on the ‘gcc’ build. For the AOCC build, you can replace ‘gcc’ with ‘aocc’.

**Installing Debian Package**

Complete the following steps to install the AOCL Debian package:
1. Download the AOCL 3.1 Debian package to the target machine.

2. Check the installation path before installing.
   ```bash
   $ dpkg -c aocl-linux-gcc-3.1.0_1_amd64.deb
   
   Or
   $ sudo dpkg -i aocl-linux-gcc-3.1.0_1_amd64.deb
   3. Install the package.
   ```
   ```bash
   $ sudo dpkg -i aocl-linux-gcc-3.1.0_1_amd64.deb
   Or
   $ sudo apt install ./aocl-linux-gcc-3.1.0_1_amd64.deb
   
   **Note:** You must have the sudo privileges to perform this action.
   
   4. Display the installed package information along with the package version and a short description.
   ```bash
   $ dpkg -s aocl-linux-gcc-3.1.0
   $ dpkg -L aocl-linux-gcc-3.1.0
   
   5. List the contents of the package.
   ```
   ```bash
   $ dpkg -L aocl-linux-gcc-3.1.0
   
   6. AOCL 3.1 supports AOCL libraries with the following two integer types:
   - LP64 libraries and header files are installed in `/INSTALL_PATH/lib_LP64` and `/INSTALL_PATH/include_LP64` respectively.
   - ILP64 libraries and header files are installed in `/INSTALL_PATH/lib_ILP64` and `/INSTALL_PATH/include_ILP64` respectively.

   By default, LP64 libraries and header files are available in `/INSTALL_PATH/lib` and `/INSTALL_PATH/include` respectively.

   Where,
   - `INSTALL_PATH`: `/opt/AMD/aocl/aocl-linux-<compiler>-3.1.0`
   - Compiler: aocc or gcc

   For example, `INSTALL_PATH` for aocc compiler is `/opt/AMD/aocl/aocl-linux-aocc-3.1.0/`

   7. To change the default library path to ILP64 / LP64, use the script as follows:
   ```bash
   cd /opt/AMD/aocl/aocl-linux-<compiler>-3.1.0/
   sudo bash setenv_aocl.sh <ilp64 / lp64>
   
   Uninstalling Debian package
   
   Execute one of the following commands to uninstall the AOCL Debian package:
   ```
   ```bash
   $ sudo dpkg -r aocl-linux-gcc-3.1.0
   Or
   $ sudo apt remove aocl-linux-gcc-3.1.0
   
   Installing RPM Package
   
   Complete the following steps to install the AOCL RPM package:
   ```
   1. Download the AOCL 3.1 RPM package to the target machine.
   2. Install the package.
   ```
   ```bash
   $ sudo rpm -ivh aocl-linux-gcc-3.1.0-1.x86_64.rpm
3. Display the installed package information along with the package version and a short description.

$ rpm -qi aocl-linux-gcc-3.1.0-1.x86_64

4. List the contents of the package.

rpm -ql aocl-linux-gcc-3.1.0-1

5. AOCL 3.1 supports AOCL libraries with the following two integer types:

- LP64 libraries and header files are installed in /INSTALL_PATH/lib_LP64 and /INSTALL_PATH/include_LP64 respectively.
- ILP64 libraries and header files are installed in /INSTALL_PATH/lib_ILP64 and /INSTALL_PATH/include_ILP64 respectively.

By default, LP64 libraries and header files are available in /INSTALL_PATH/lib and /INSTALL_PATH/include respectively.

Where,

- INSTALL_PATH: /opt/AMD/aocl/aocl-linux-<compiler>-3.1.0/
- Compiler: aocc or gcc

For example, INSTALL_PATH for aocc compiler is /opt/AMD/aocl/aocl-linux-aocc-3.1.0/.

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

```
cd /opt/AMD/aocl/aocl-linux-<compiler>-3.1.0/
sudo bash setenv_aocl.sh <ilp64 / lp64>
```

**Uninstalling RPM package**

Execute the following command to uninstall the AOCL RPM package:

```
$ rpm -e aocl-linux-gcc-3.1.0-1
```

### 3.2.4 Using Windows Packages

#### Installing a Windows Package

Complete the following steps to install the AOCL Windows package:


2. Double-click the executable.
   
   The installation wizard is displayed.

3. Click the **Next** button.

4. Accept the **License Agreement** and click the **Next** button.

5. Select the libraries to be installed and the destination folder.

6. Click the **Install** button to begin the installation.
7. Click the **Finish** button to complete the installation.

**Uninstalling a Windows Package**

Complete the following steps to uninstall the AOCL Windows binaries:

1. Double-click the AOCL Windows installer.
2. Click the **Remove** button.
   
   Alternatively, you can also use the Add or remove programs option in Windows.
3. Click the **Finish** button to complete the uninstallation.
Chapter 4    BLIS Library for AMD

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

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

4.1    Installation on Linux

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

4.1.1    Build BLIS from Source

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

You can use the following ways to build BLIS using the configure/make method:

• **auto** — This configuration generates a binary optimized for the build machine’s AMD “Zen” core architecture. This is useful when you build the library on the target system. Starting from the BLIS 2.1 release, the *auto* configuration option enables selecting the appropriate build configuration based on the target CPU architecture. For example, for a build machine using the 1st Gen AMD EPYC™ (code name “Naples”) processor, the *zen* configuration will be auto-selected. For a build machine using the 2nd Gen AMD EPYC™ processor (code name "Rome"), the *zen2* configuration will be auto-selected. From BLIS 3.0 forward, *zen3* will be auto-selected for the 3rd Gen AMD EPYC™ processor (code name "Milan").

• **zen** — This configuration generates a binary compatible with AMD “Zen” architecture and is optimized for it. The architecture of the build machine is not relevant.

• **zen2** — This configuration generates binary compatible with AMD “Zen2” architecture and is optimized for it. The architecture of the build machine is not relevant.

• **zen3** — This configuration generates binary compatible with AMD “Zen3” architecture and is optimized for it. The architecture of the build machine is not relevant.

• **amdzen** — The library built using this configuration generates a binary compatible with and optimized for AMD “Zen”, AMD “Zen2”, and AMD “Zen3” architectures. The architecture of
the build machine is not relevant. The architecture of the target machine is checked during the runtime, based on which, the relevant optimizations are picked up automatically.

This feature is also called Dynamic Dispatch. For more information, refer “Dynamic Dispatch” on page 29.

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

```
./configure --help.
```

4.1.1.1 Single-thread BLIS

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

2. Configure the library as required:

   **GCC (Default)**
   ```
   $ ./configure --enable-cblas --prefix=<your-install-dir> auto
   ```

   **AOCC**
   ```
   $ ./configure --enable-cblas --prefix=<your-install-dir> --complex-return=intel CC=clang CXX=clang++ auto
   ```

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

4.1.1.2 Multi-thread BLIS

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

2. Configure the library as required:

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

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

   

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

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

For more information on multi-threaded implementation in BLIS, refer GitHub BLIS Multi-threading Documentation (https://github.com/flame/blis/blob/master/docs/Multithreading.md).
4.1.1.3 Verifying BLIS Installation

The BLIS source directory contains the test cases which demonstrate the usage of BLIS APIs. To execute the tests, navigate to the BLIS source directory and run the following command:

```
$ BLIS_NUM_THREADS=1 make check
```

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

```
$ BLIS_NUM_THREADS=1 make checkcpp
```

4.1.2 Using Pre-built Binaries

AMD optimized BLIS library binaries for Linux are available in the following URLs:

- [https://github.com/amd/blis/releases](https://github.com/amd/blis/releases)

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

The master installer includes the following:

- Single threaded and multi-threaded BLIS binaries.
- Binaries built with `amdzen` config with LP64 and ILP64 integer support.
- Multi-thread BLIS binary (libblis-mt) 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 Application Development Using BLIS

This section explains how to call BLIS APIs, compile, and link with the BLIS library.

4.2.1 API Compatibility Layers (Calling BLIS)

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

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

BLIS has its own APIs which provides more flexibility and control to achieve the best performance.
The following table lists all the supported layers:

**Table 1. BLIS API Compatibility Layers**

<table>
<thead>
<tr>
<th>API Compatibility Layer</th>
<th>Header Files</th>
<th>Configuration Option</th>
<th>Usages</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLAS (Fortran)</td>
<td>Not applicable</td>
<td>--enable-blas</td>
<td>Use this option when calling BLIS from Fortran applications. API Name Format: DGEMM</td>
</tr>
<tr>
<td>BLAS (C)</td>
<td>blis.h</td>
<td>--enable-blas</td>
<td>Use this option when calling BLIS from C application using BLAS type parameters. API Name Format: dgemm_</td>
</tr>
<tr>
<td>CBLAS</td>
<td>cblas.h</td>
<td>--enable-cblas (Implies --enable-blas)</td>
<td>Use this option when calling BLIS from C application using CBLAS type parameters. API Name Format: cblas_dgemm</td>
</tr>
<tr>
<td>BLIS - C Non Standard</td>
<td>blis.h</td>
<td>Default</td>
<td>This is BLIS library specific (non-standard) interface, it provides most flexibility in calling BLIS for best performance. However, these applications will not be portable to other BLAS/CBLAS compatible libraries. API Name Format: bli_gemm API Name Format: blis_gemm_ex</td>
</tr>
<tr>
<td>BLIS – CPP Non Standard</td>
<td>blis.hh</td>
<td>Default</td>
<td>This is BLIS library specific (non-standard) C++ interface. This interface follows same parameter order as CBLAS. However, these applications will not be portable to other BLAS/CBLAS compatible libraries. API Name Format: blis::gemm</td>
</tr>
</tbody>
</table>
4.2.2 API Compatibility - Advance Options

The API compatibility can be further extended to meet additional requirements for input sizes (ILP64) and different ways in which complex numbers are handled. The following table explains such options:

Table 2. BLIS API Compatibility - Advance Options

<table>
<thead>
<tr>
<th>Feature</th>
<th>Configuration Option</th>
<th>Usages</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILP64 Support</td>
<td>--blas-int-size=SIZE</td>
<td>This option can be used to specify the integer types used in external BLAS/CBLAS interfaces. Accepted Values: ILP64 - SIZE = 64 LP64 - SIZE = 32 (Default)</td>
</tr>
<tr>
<td>Complex Number return handling</td>
<td>--complex-return=gnu</td>
<td>intel</td>
</tr>
</tbody>
</table>

4.2.3 Linking Application with BLIS

The BLIS library can be linked statically or dynamically with the user application. It has a separate binary for single-threaded and multi-threaded implementation.

The basic build command is as following:

gcc test_blis.c -I<path-to-BLIS-header> <link-options> -o test_blis.x

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

Table 3. BLIS Application - Link Options

<table>
<thead>
<tr>
<th>Application Type</th>
<th>Linking Type</th>
<th>Link Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-threaded</td>
<td>Static</td>
<td>&lt;path-to-BLIS-library&gt;/libblis.a -lm -lpthread</td>
</tr>
<tr>
<td>Single-threaded</td>
<td>Dynamic</td>
<td>-L&lt;path-to-BLIS-library&gt; -lblis -lm -lpthread</td>
</tr>
<tr>
<td>Multi-threaded</td>
<td>Static</td>
<td>&lt;path-to-BLIS-library&gt;/libblis-mt.a -lm -fopenmp</td>
</tr>
<tr>
<td>Multi-threaded</td>
<td>Dynamic</td>
<td>-L&lt;path-to-BLIS-library&gt; -lblis-mt -lm -fopenmp</td>
</tr>
</tbody>
</table>
4.2.3.1 Example - Dynamic Linking and Execution

BLIS can be built as a shared library. By default, the library is built as both static and shared libraries. Complete the following steps to build a shared lib version of BLIS and link it with the user application:

1. During configuration, enable the support for the shared lib using the following command:
   ```
   ./configure --disable-static --enable-shared zen
   ```

2. Link the application with the generated shared library using the following command:
   ```
   gcc CBLAS_DGEMM_usage.c -I path/to/include/blis/ -L path/to/libblis.so -lblis -lm -lpthread -o CBLAS_DGEMM_usage.x
   ```

3. Ensure that the shared library is available in the library load path. Run the application using the following command (for this demo we will use the `BLAS_DGEMM_usage.c`):
   ```
   $ export LD_LIBRARY_PATH="path/to/libblis.so"
   $ ./BLAS_DGEMM_usage.x
   a =
   1.000000        2.000000
   3.000000        4.000000
   b =
   5.000000        6.000000
   7.000000        8.000000
   c =
   19.000000       22.000000
   43.000000       50.000000
   ```

4.2.4 Example Application - BLIS Usage in FORTRAN

BLIS can be used with the FORTRAN applications through the standard BLAS API.
For example, the following FORTRAN code does a double precision general matrix-matrix multiplication. It calls the 'DGEMM' BLAS API function to accomplish this. A sample command to compile and link it with the BLIS library is shown in the following code:

```fortran
! File: BLAS_DGEMM_usage.f
!
program dgemm_usage

implicit none

EXTERNAL DGEMM

DOUBLE PRECISION, ALLOCATABLE :: a(:,,:)
DOUBLE PRECISION, ALLOCATABLE :: b(:,,:)
DOUBLE PRECISION, ALLOCATABLE :: c(:,,:)
INTEGER I, J, M, N, K, lda, ldb, ldc
DOUBLE PRECISION alpha, beta

M=2
N=M
K=M
lda=M
ldb=K
ldc=M
alpha=1.0
beta=0.0

ALLOCATE(a(lda,K), b(ldb,N), c(ldc,N))

a=RESHAPE((/ 1.0, 3.0, &
            2.0, 4.0  /), &
            (/lda,K/))
b=RESHAPE((/ 5.0, 7.0, &
            6.0, 8.0  /), &
            (/ldb,N/))

WRITE(*,*) ("a =")
DO I = LBOUND(a,1), UBOUND(a,1)
    WRITE(*,*) (a(I,J), J=LBOUND(a,2), UBOUND(a,2))
END DO
WRITE(*,*) ("b =")
DO I = LBOUND(b,1), UBOUND(b,1)
    WRITE(*,*) (b(I,J), J=LBOUND(b,2), UBOUND(b,2))
END DO
CALL DGEMM('N','N',M,N,K,alpha,a,lda,b,ldb,beta,c,ldc)

WRITE(*,*) ("c =")
DO I = LBOUND(c,1), UBOUND(c,1)
    WRITE(*,*) (c(I,J), J=LBOUND(c,2), UBOUND(c,2))
END DO

end program dgemm_usage
```
A sample compilation command with gfortran compiler for the code above:

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

### 4.2.5 BLIS Usage in C

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

#### 4.2.5.1 Example Application - Using BLIS with BLAS API in C

Following is the C version of the FORTRAN code in section 4.2.4. It uses the standard BLAS API. The following process takes place during the execution of the code:

1. The matrices are transposed to account for the row-major storage of C and the column-major convention of BLAS (inherited from FORTRAN).
2. The function arguments are passed by address again to be in line with FORTRAN conventions.
3. There is a trailing underscore in the function name ('dgemm_') as BLIS' BLAS APIs require FORTRAN compilers to add a trailing underscore.
4. "blis.h" is included as a header. A sample command to compile it and link with the BLIS library is also shown in the following code:

```c
// File: BLAS_DGEMM_usage.c
// Example code to demonstrate BLAS DGEMM usage

#include<stdio.h>
#include "blis.h"

#define DIM 2

int main() {

    double a[DIM * DIM] = { 1.0, 3.0, 2.0, 4.0 };  
    double b[DIM * DIM] = { 5.0, 7.0, 6.0, 8.0 };  
    double c[DIM * DIM]; 
    int I, J, M, N, K, lda, ldb, ldc; 
    double alpha, beta; 

    M = DIM; 
    N = M; 
    K = M; 
    lda = M; 
    ldb = K; 
    ldc = M; 
    alpha = 1.0; 
    beta = 0.0; 

    printf("a = \n");
    for ( I = 0; I < M; I ++ ) {
        for ( J = 0; J < K; J ++ ) {
            printf("%f\t", a[J * K + I]);
        }
        printf("\n");
    }

    printf("b = \n");
    for ( I = 0; I < K; I ++ ) {
        for ( J = 0; J < N; J ++ ) {
            printf("%f\t", b[J * N + I]);
        }
        printf("\n");
    }

dgemm_("N","N", &M, &N, &K, &alpha, a, &lda, b, &ldb, &beta, c, &ldc);

    printf("c = \n");
    for ( I = 0; I < M; I ++ ) {
        for ( J = 0; J < N; J ++ ) {
            printf("%f\t", c[J * N + I]);
        }
        printf("\n");
    }

    return 0;
}
```
A sample compilation command with a gcc compiler for the code above:

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

### 4.2.5.2 Example Application - Using BLIS with CBLAS API

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

The following process takes place during the execution of the code:

1. CBLAS Layout option allows you to choose between row-major and column-major layouts (row-major layout is used in the example, which is in line with C-style).
2. The function arguments can be passed by the value too.
3. "cblas.h" is included as a header. A sample command to compile it and link with the BLIS library is also shown in the following code:

```c
// File: CBLAS_DGEMM_usage.c
// Example code to demonstrate CBLAS DGEMM usage
#include<stdio.h>
#include "cblas.h"

#define DIM 2

int main() {
    double a[DIM * DIM] = { 1.0, 2.0, 3.0, 4.0 };
    double b[DIM * DIM] = { 5.0, 6.0, 7.0, 8.0 };
    double c[DIM * DIM];
    int I, J, M, N, K, lda, ldb, ldc;
    double alpha, beta;

    M = DIM;
    N = M;
    K = M;
    lda = M;
    ldb = K;
    ldc = M;
    alpha = 1.0;
    beta = 0.0;

    printf("a = \n");
    for ( I = 0; I < M; I ++ ) {
        for ( J = 0; J < K; J ++ ) {
            printf("%f\t", a[I * K + J]);
        }
        printf("\n");
    }

    printf("b = \n");
    for ( I = 0; I < K; I ++ ) {
        for ( J = 0; J < N; J ++ ) {
            printf("%f\t", b[I * N + J]);
        }
        printf("\n");
    }

    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, M, N, K, alpha, a, lda, b, ldb, beta, c, ldc);

    printf("c = \n");
    for ( I = 0; I < M; I ++ ) {
        for ( J = 0; J < N; J ++ ) {
            printf("%f\t", c[I * N + J]);
        }
        printf("\n");
    }

    return 0;
}
```
**Note:** To get the CBLAS API with BLIS, you must provide the flag `--enable-cblas` to the `configure` command while building the BLIS library.

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

```bash
gcc CBLAS_DGEMM_usage.c -I/path/to/include/blis/ -lpthread path/to/libblis.a
```

### 4.2.5.3 Returning Complex Number

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

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

**gfortran Example:**

- Configure Option:
  ```bash
  --complex-return=gnu
  ```
- API Call:
  ```c
  ret_value = cdotc_(&n, x, &incx, y, &incy);
  ```

**ifort example:**

- Configure Option:
  ```bash
  --complex-return=intel
  ```
- API Call:
  ```c
  cdotc_(&ret_value, &n, x, &incx, y, &incy);
  ```

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

### 4.3 Migrating/Porting

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

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

1. Update the source code to include the correct header files.
2. Update the build script or makefile to use correct compile or link option.
The following table lists the compile and linker options while porting to BLIS:

Table 4. Porting to BLIS

<table>
<thead>
<tr>
<th></th>
<th>MKL</th>
<th>OpenBLAS</th>
<th>BLIS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>mkl.h</td>
<td>cblas.h</td>
<td>blis.h/cblas.h</td>
</tr>
<tr>
<td><strong>Header File</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Link Options</strong></td>
<td>-lmkl_intel_lp64</td>
<td>-lopenblas</td>
<td>-lm -lblis</td>
</tr>
<tr>
<td></td>
<td>-lmkl_core</td>
<td></td>
<td>-lm -fopenmp</td>
</tr>
<tr>
<td></td>
<td>-lmkl_blacs_intelmpi_ilp64</td>
<td></td>
<td>-lblis-mt</td>
</tr>
<tr>
<td></td>
<td>-lmkl_intel_thread</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 4.4 Using BLIS Library Features

#### 4.4.1 Dynamic Dispatch

Starting from AOCL 3.1, BLIS supports Dynamic Dispatch feature. It enables you to use the same binary on different architectures.

**4.4.1.1 Purpose**

Before Dynamic Dispatch, the user had to build different binaries for each CPU architecture, that is, AMD “Zen”, AMD “Zen2”, and AMD “Zen3” architectures. Furthermore, when building the application, users had to ensure that they used the correct AMD “Zen”-based library as needed for the platform. This becomes challenging when using BLIS on a cluster having nodes of different architectures.

Dynamic Dispatch addresses this issue by building a single binary containing a support for all the AMD “Zen” architectures. At the runtime, the Dynamic Dispatch feature enables optimizations specific to the detected AMD “Zen” architecture.

**4.4.1.2 On non-AMD “Zen” Architectures**

The Dynamic Dispatch feature supports AMD “Zen”, AMD “Zen2”, and AMD “Zen3” architectures in a single binary. However, it also includes the support for standard x86 architecture. The generic architecture uses a pure C implementation of the APIs and does not use any architecture-specific features.

The specific compiler flags used for building the library with generic configuration are:

-02 -funsafe-math-optimizations -ffp-contract=fast -Wall -Wno-unused-function -Wfatal-errors

**Note:** As no architecture specific optimization and vectorized kernels are enabled, performance with the generic architecture may be significantly lower than the architecture-specific implementation.
4.4.1.3 Using Dynamic Dispatch

Building BLIS

Dynamic Dispatch must be enabled while building the BLIS library. This is done by building the library for **amdzen** configuration as explained in “Build BLIS from Source” on page 17.

Architecture Selection at Runtime

For most of the use cases, Dynamic Dispatch will detect the underlying architecture and enable appropriate code paths and optimizations. However, for debugging, BLIS can be forced to use specific architecture by setting environment variable `BLIS_ARCH_TYPE` as follows:

```
$ BLIS_ARCH_TYPE=value <BLIS linked application>
value = {6 – zen3, 7 – zen2, 8 – zen, 22 – generic}
```

Dynamic Dispatch can print addition debugging information which can be enabled by setting the environment `BLIS_ARCH_DEBUG = 1`.

4.4.2 BLIS - Running in-built Test Suite

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

For more information, refer [https://github.com/flame/blis/blob/master/docs/Testsuite.md](https://github.com/flame/blis/blob/master/docs/Testsuite.md).

4.4.2.1 Multi-thread Test Suite Performance

Starting from BLIS 3.1, the dynamic selection of number of threads is supported. If the number of threads are not specified, BLIS uses the maximum number of threads possible on the system. A higher number of threads result in better performance for medium to large size matrices found in practical use cases.

However, the higher number of threads results in poor performance for very small sizes used by the test and check features. Hence, you must specify the number of threads while running the test/test suite.

The recommended number of threads to run the test suite is 1 or 2.

Running Test Suite

Execute the following command to invoke the test suite:

```
$ BLIS_NUM_THREADS=2 make test
```
The sample output from the execution of the command is as follows:

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

### 4.4.3 Testing/Benchmarking

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

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

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

**Enabling File Inputs**

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

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

**Building Test Driver**

Execute the following commands to build the test driver:

```
$ cd tests
$ make blis
```
Creating an Input File

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

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

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

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

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

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

Running the Tests

Execute the following commands to run the tests:

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

An execution sample (with the test driver) for GEMM is as follows:

```bash
$ cat inputs.txt
200 100 100 200 200 200
10 4 1 100 100 100
4000 4000 400 4000 4000 4000
$ ./test_gemm_blis.x inputs.txt outputs.txt
```

```
~~~~~~~~~~ BLAS  m   k   n   cs_a   cs_b   cs_c   gflops
data_gemm_blis  200  100  100     200     200     200   27.211
data_gemm_blis   10     4     1     100     100     100   0.027
data_gemm_blis  4000  4000  400    4000    4000    4000   45.279
```

```bash
$ cat outputs.txt
m   k   n   cs_a   cs_b   cs_c   gflops
200  100  100     200     200     200   27.211
10    4     1     100     100     100   0.027
4000  4000  400    4000    4000    4000   45.279
```
4.4.4 BLIS APIs

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

<table>
<thead>
<tr>
<th>API</th>
<th>Usages</th>
<th>Comments/Caveats</th>
</tr>
</thead>
<tbody>
<tr>
<td>bli_info_get_version_str</td>
<td>Gets the version info of the running BLIS library.</td>
<td>Returns the version string in the form of “AOCL-3.1”.</td>
</tr>
<tr>
<td>bli_info_get_enable_openmp</td>
<td>Returns if openmp/pthread is enabled or disabled.</td>
<td>Returns true/false.</td>
</tr>
<tr>
<td>bli_info_get_enable_pthreads</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bli_info_get_enable_threading</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bli_thread_get_num_threads</td>
<td>Returns the global number of threads.</td>
<td>Returns the number of threads per operation.</td>
</tr>
<tr>
<td>bli_thread_set_num_threads</td>
<td>Sets the global number of threads.</td>
<td>Sets the number of threads for the subsequent BLAS calls.</td>
</tr>
<tr>
<td>bli_thread_set_ways</td>
<td>Sets the number of threads for different levels of parallelization.</td>
<td>Using this API, you can specify the number of threads used for the different loops.</td>
</tr>
</tbody>
</table>

Notes:
1. Refer https://github.com/flame/blis/blob/master/docs/Multithreading.md#specifying-multithreading

4.5 Debugging and Troubleshooting

4.5.1 Debugging Build Using GDB

The BLIS library can be debugged on Linux using GDB. To enable the debugging support, build the library with the --enable-debug flag. Use following commands to configure and build the debug version of BLIS:

```bash
$ cd blis_src
$ ./configure --enable-cblas --enable-debug auto
$ make -j
```

Use the following commands to link the application with the binary and build application with debug support:

```bash
$ cd blis_src
$ gcc -g -O0 -lpthread -I<path-to-BLIS-header> <path-to-BLIS-library>/libblis.a test_gemm.c -o test_gemm_blis.x
```
You can debug the application using gdb. A sample output of the gdb session is as follows:

```bash
$ gdb ./test_gemm_blis.x
GNU gdb (GDB) Red Hat Enterprise Linux 8.2-12.el8
.. .. ..
Reading symbols from ./test_gemm_blis.x...done.
(gdb) break bli_gemm_small
Breakpoint 1 at 0x677543: file kernels/zen/3/bli_gemm_small.c, line 110.
(gdb) run
Starting program: /home/dipal/work/blis_dtl/test/test_gemm_blis.x
Using host libthread_db library "/lib64/libthread_db.so.1".
BLIS Library version is : AOCL BLIS 3.1

Breakpoint 1, bli_gemm_small (alpha=0x7fffffffcf40, a=0x2471b30, b=0x7fffffffd1c0, beta=0x2465400 <BLIS_ZERO>, c=0x4fe66e <bli_obj_equals+300>, cntx=0x7fffffffb320, cntl=0x0) at kernels/zen/3/bli_gemm_small.c:110
110  {
(gdb) bt
#0  bli_gemm_small (alpha=0x7fffffffcf40, a=0x2471b30, b=0x7fffffffd1c0, beta=0x2465400
<BLIS_ZERO>, c=0x4fe66e <bli_obj_equals+300>, cntx=0x7fffffffb320, cntl=0x0) at kernels/zen/3/bli_gemm_small.c:110
#1  0x000000000007caab6 in bli_gemm_front (alpha=0x7fffffffd1c0, a=0x7fffffffd120, b=0x7fffffffd000, beta=0x7fffffffcf40, c=0x7fffffffcf40, cntx=0x2471b30, rntm=0x7fffffffcf50, cntl=0x0) at frame/3/gemm/bli_gemm_front.c:83
#2  0x00000000005badf42 in bli_gemmnat (alpha=0x7fffffffd1c0, a=0x7fffffffd120, b=0x7fffffffd000, beta=0x7fffffffcf40, c=0x7fffffffcf40, cntx=0x2471b30, rntm=0x7fffffffcf50) at frame/ind/oapi/bli_l3_nat_oapi.c:83
#3  0x00000000005474a2 in dgemm_ (transa=0x7fffffffd363 "N\320\a", transb=0x7fffffffd362 "NN\320\a", m=0x7fffffffd36c, n=0x7fffffffd364, k=0x7fffffffd368, alpha=0x24733c0, a=0x7fffffffd378, lda=0x7fffffffd378, b=0x7fffffffd374, ldb=0x7fffffffd374, beta=0x2473340, c=0x7fffffffd370, ldc=0x7fffffffd370) at frame/compat/bla_gemm.c:559
#4  0x0000000000413a1c in main (argc=1, argv=0x7fffffffd988) at test_gemm.c:321
(gdb)
```

### 4.5.2 Viewing Logs

The AMD BLIS library provides inbuilt Debug and Trace features:

- **Trace Log** identifies the code path taken in terms of function call chain. It prints the information on the functions invoked and their order.

- **Debug Log** prints the other debugging information, such as values of input parameters, content, and data structures.

The key features of this functionality are as follows:

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

### 4.5.2.1 Function Call Tracing

The function call tracing is implemented using hard instrumentation of the BLIS code. Here, the functions are grouped as per their position in the call stack. You can configure the level up to which the traces must be generated.

Complete the following steps to enable and view the traces:

1. Enable the trace support as follows:
   a. Modify the source code to enable tracing.
      
      ```
      Open file <blis folder>/aocl_dtl/aocldtlcf.h
      ``
   
   b. Change the following macro from 0 to 1:
      ```
      #define AOCL_DTL_TRACE_ENABLE       0
      ```

2. Configure the trace depth level.
   a. Modify the source code to specify the trace depth level.
      ```
      Open file <blis folder>/aocl_dtl/aocldtlcf.h
      ```
   
   b. Change the following macro as required. Beginning with Level 5 should be a good compromise in terms of details and resource requirement. The higher the level, the deeper is the call stack. A lower level reduces the depth of the call stack used for a trace generation.
      ```
      #define AOCL_DTL_TRACE_LEVEL  AOCL_DTL_LEVEL_TRACE_5
      ```

3. Build the library as explained in “Build BLIS from Source” on page 17.
4. Run the application to generate the trace data.

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

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

Figure 1. Sample Run of Function Call Tracing

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

\[ P<\text{process id}>_T<\text{thread id}>_{\text{aocldtl_trace}}.txt \]

5. View the trace data.

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

4.5.2.2 Debug Logging

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

Complete the following steps to enable and view the debug logs:

1. Enable the debug log support as follows:
   a. Modify the source code to enable debug logging.

   ```bash
   Open file <blis folder>/aocl_dtl/aocldtlcf.h
   ```
b. Change the following macro from 0 to 1:

```c
#define AOCL_DTL_LOG_ENABLE 0
```

2. Configure the trace depth level.

   a. Modify the source code to specify the debug log depth level.

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

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

   ```c
   #define AOCL_DTL_TRACE_LEVEL AOCL_DTL_LEVEL_TRACE_5
   ```

3. Build the library as explained in “Build BLIS from Source” on page 17.

4. Run the application to generate the trace data.

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

   The following figure shows a sample running of BLIS with the debug logs enabled using the test_gemm application:

   ![Sample Run with Debug Logs Enabled](image)

   **Figure 2. Sample Run with Debug Logs Enabled**

   The debug logs for each thread are saved in the file with appropriate naming conventions. The `.txt` extension is used to signify the readable file:

   ```
P<process id>_T<thread id>_aocldtl_log.txt
   ```
5. View the debug logs.

The output of the debug logs is in a readable format, you can open the file in any of the text editors. The following figure shows the sample output for one of the threads of test_gemm application:

![Debug Logs Showing Input Values of GEMM](image)

Figure 3. Debug Logs Showing Input Values of GEMM

### 4.5.3 Usages and Limitations

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

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

### 4.6 Build BLIS from Source on Windows

GitHub URL: [https://github.com/amd/blis](https://github.com/amd/blis)

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

**Prerequisites**

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

  Figure 4. Microsoft Visual Studio Prerequisites

4.6.1 Building BLIS using GUI

4.6.1.1 Preparing Project with CMake GUI

Complete the following steps in the CMake GUI:
1. Set the **source** (folder containing BLIS source code) and **build** (folder in which the project files will be generated, for example, **out**) folder paths as shown in the following figure:

![CMake Source and Build Folders](image)

**Figure 5.** CMake Source and Build Folders

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

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

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

![Set Generator and Compiler](image)

**Figure 6.** Set Generator and Compiler
4. Update the options based on the project requirements. All the available options are listed in the following table:

**Table 6. CMake Config Options**

<table>
<thead>
<tr>
<th>Feature</th>
<th>CMake Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD CPU architecture</td>
<td>AOCL_BLIS_FAMILY:STRING=zen/zen2/zen3</td>
</tr>
<tr>
<td>Enable verbose mode</td>
<td>ENABLE_VERBOSE=ON</td>
</tr>
<tr>
<td>Shared library</td>
<td>BUILD_SHARED_LIBS=ON</td>
</tr>
<tr>
<td>Static library</td>
<td>BUILD_SHARED_LIBS=OFF</td>
</tr>
<tr>
<td>Debug/Release build type</td>
<td>CMAKE_BUILD_TYPE=Debug/Release</td>
</tr>
<tr>
<td>Dynamic Dispatcher</td>
<td>AOCL_BLIS_FAMILY:STRING=amdzen</td>
</tr>
<tr>
<td>Enable single threading</td>
<td>ENABLE_MULTITHREADING=OFF</td>
</tr>
<tr>
<td>Enable multi-threading with OpenMP and AOCL dynamic enabled</td>
<td>ENABLE_MULTITHREADING=ON</td>
</tr>
<tr>
<td>Enable multi-threading with OpenMP and AOCL dynamic disabled</td>
<td>ENABLE_MULTITHREADING=OFF</td>
</tr>
<tr>
<td>Enable BLAS/CBLAS support</td>
<td>ENABLE_BLAS=ON</td>
</tr>
<tr>
<td>Enable 32-bit BLIS/BLAS integer size</td>
<td>BLIS_ENABLE_ILP64=OFF</td>
</tr>
<tr>
<td>Enable 64-bit BLIS/BLAS integer size</td>
<td>BLIS_ENABLE_ILP64=ON</td>
</tr>
<tr>
<td>Flags that are enabled by default</td>
<td>ENABLE_JRIR_SLAB</td>
</tr>
<tr>
<td></td>
<td>ENABLE_PBA_POOLS</td>
</tr>
<tr>
<td></td>
<td>ENABLE_SBA_POOLS</td>
</tr>
<tr>
<td></td>
<td>ENABLE_MIXED_DT</td>
</tr>
<tr>
<td></td>
<td>ENABLE_MIXED_DT_EXTRA_MEM</td>
</tr>
<tr>
<td></td>
<td>ENABLE_SUP_HANDLING</td>
</tr>
<tr>
<td></td>
<td>ENABLE_PRAGMA_OMP_SIMD</td>
</tr>
<tr>
<td>Flags that are disabled by default</td>
<td>ENABLE_JRIR_RR</td>
</tr>
<tr>
<td></td>
<td>ENABLE_MEM_TRACING</td>
</tr>
<tr>
<td></td>
<td>ENABLE_MEMKIND</td>
</tr>
<tr>
<td></td>
<td>ENABLE_SANDBOX</td>
</tr>
<tr>
<td>Use APIs without trailing underscore</td>
<td>ENABLE_NO_UNDERSCORE_API</td>
</tr>
</tbody>
</table>
4.6.1.2 Building the Project in Visual Studio GUI

Complete the following steps in the Microsoft Visual Studio GUI:

1. Open the project generated by CMake (build folder) in “Preparing Project with CMake GUI” on page 39.

2. To generate BLIS binaries, build the **AOCL-LibBlis-Win** project.

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

   For example, `blis/bin/Release/AOCL-LibBlis-Win.dll` or `AOCL-LibBlis-Win.lib`
4.6.2 Building BLIS using Command-line Arguments

The project configuration and build procedures can be triggered from the command prompt as well. The corresponding steps are described in the following sections.

4.6.2.1 Configuring the Project in Command Prompt

In the BLIS project folder, create a folder `out`. Open the command prompt in this directory and run the following command to configure the project:

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

You can refer Table 6 and update the parameter options in the command according to the project requirements.

4.6.2.2 Building the Project in Command Prompt

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

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

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

4.6.3 Building and Running Test Suite

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

4.6.3.1 Running Individual Tests

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

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

4.6.3.2 Running Test Suite

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

```bash
Release> test_libblis.exe
```
4.6.3.3 Running Multi-thread Tests

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

1. Copy the relevant input files for the tests from \blis\testsuite or \blis\bench to the \blis\bin\release folder.

2. Copy libomp.lib and libomp.dll respectively from the Microsoft Visual Studio folders \VC\Tools\Llvm\lib and \VC\Tools\Llvm\bin to the \blis\bin\release folder.

3. Set the threading environment variables in the same command prompt session as the test runs.

   For example:
   
   Release> set BLIS_NUM_THREADS=x  (x could be no of threads)
   Release> set OMP_PROC_BIND=spread
   Release> TestGemm.exe inputgemm.txt output.txt
Chapter 5  libFLAME Library for AMD

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

Starting from AOCL 3.1 release, AMD optimized version of libFLAME is compatible with LAPACK 3.10.0 specification. In combination with the BLIS library, which includes optimizations for the AMD EPYC™ processor family, libFLAME enables running high performing LAPACK functionalities on AMD platforms. The AMD version of libFLAME supports C, FORTRAN, and C++ template interfaces for the LAPACK functionalities.

5.1  Installing on Linux

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

5.1.1  Building libFLAME from Source

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

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

Prerequisites

The following prerequisites must be met for installing libFLAME:

- Python must be installed on the target machine.
- Make version 4.x or later must be installed.

Build Steps

Complete the following steps to build libFLAME from source:

1. Clone the Git repository (https://github.com/amd/libflame.git).
2. Run the configure script. An example below shows a few sample options. Enable/disable the other flags as required.
   - With GCC (default)
     **Using 32-bit Integer (LP64)**
     
     ```
     $ ./configure --enable-blas-ext-gemmt --enable-lapack2flame --enable-external-lapack-
     interfaces --enable-dynamic-build --enable-max-arg-list-hack --prefix=<your-install-
     dir>
     ```
     
     **Using 64-bit Integer (ILP64)**
     
     ```
     $ ./configure --enable-blas-ext-gemmt --enable-lapack2flame --enable-external-lapack-
     interfaces --enable-dynamic-build --enable-max-arg-list-hack --prefix=<your-install-dir>
     ```
     
     - With AOCC
     
     ```
     $ export CC=clang
     $ export CXX=clang++
     $ export FC=flang
     $ export FLIBS="-lflang"
     ```
     
     **Using 32-bit Integer (LP64)**
     
     ```
     $ ./configure --enable-blas-ext-gemmt --enable-lapack2flame --enable-external-lapack-
     interfaces --enable-dynamic-build --enable-max-arg-list-hack --enable-f2c-dotc --
     enable-void-return-complex --prefix=<your-install-dir>
     ```
     
     **Using 64-bit Integer (ILP64)**
     
     ```
     $ ./configure --enable-blas-ext-gemmt --enable-lapack2flame --enable-external-lapack-
     interfaces --enable-dynamic-build --enable-max-arg-list-hack --enable-f2c-dotc --
     enable-void-return-complex --prefix=<your-install-dir>
     ```

     **Note:** The option `--enable-blas-ext-gemmt` can be enabled only for BLAS libraries that support
     xGEMMT (GEMM where only upper or lower triangular part of result matrix is updated)
     APIs. It is recommended to use this option while running AMD BLIS on AMD EPYC
     processors.

3. Make and install using the following commands:
   
   ```
   $ make -j
   $ make install
   ```
   
   By default, without the configure option **prefix**, the library will be installed in `$HOME/flame`.

**5.1.2 Using Pre-built Libraries**

You can find the AMD optimized libFLAME library binaries for Linux in the following URLs:

- [https://github.com/amd/libflame/releases](https://github.com/amd/libflame/releases)
Also, libFLAME binary can be installed from the AOCL master installer tar file available in the following URL:

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

The tar file includes pre-built binaries of the other AMD libraries BLIS, LibM, FFTW, SeaLAPACK, aocl-sparse, RNG, and AMD Secure RNG.

5.2 Usage

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

5.2.1 Source Directory

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

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

5.2.1.1 Examples

• Using a single thread BLIS
  
  $ make check LIBBLAS=/home/user/aoc1/amd/3.x/libs/libblis.a

• Using a multi-thread BLIS
  
  $ BLIS_NUM_THREADS=1 make check LIBBLAS="-fopenmp /home/user/aocl/amd/3.x/libs/libblis-mt.a"

5.2.2 Use by Applications

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

5.2.2.1 Examples

• With a static library

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

• With a dynamic library

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

5.3 Building libFLAME from Source on Windows

libFLAME (https://github.com/amd/libflame) uses CMake along with Microsoft Visual Studio for building binaries from the sources on Windows. The following sections explain the GUI and command-line schemes of building the binaries and test suite.
Prerequisites

For more information, refer to the Prerequisites sub-section in section "Build BLIS from Source on Windows" on page 38.

5.3.1 Building libFLAME Using GUI

5.3.1.1 Preparing Project with CMake GUI

Complete the following steps in the CMake GUI:

1. Set the source (folder containing libFLAME source code) and build (folder in which the project files will be generated, for example, out) folder paths. It is not recommended to use the folder named build as it is already used for Linux build system.

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

3. Set the generator to Visual Studio 16 2019 and the compiler to ClangCl or LLVM.

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

Table 7. libFLAME Config Options

<table>
<thead>
<tr>
<th>Feature</th>
<th>CMake Parameter(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared library</td>
<td>BUILD_SHARED_LIBS=ON</td>
</tr>
<tr>
<td>Static library</td>
<td>BUILD_SHARED_LIBS=OFF</td>
</tr>
<tr>
<td>Flags enabled by default</td>
<td>ENABLE_BLIS1_USE_OF_FLA_MALLOC</td>
</tr>
<tr>
<td></td>
<td>ENABLE_BUILTIN_LAPACK2FLAME</td>
</tr>
<tr>
<td></td>
<td>ENABLE_EXT_LAPACK_INTERFACE</td>
</tr>
<tr>
<td></td>
<td>ENABLE_INTERNAL_ERROR_CHECKING</td>
</tr>
<tr>
<td></td>
<td>ENABLE_NON_CRITICAL_CODE</td>
</tr>
<tr>
<td></td>
<td>ENABLE_PORTABLE_TIMER</td>
</tr>
<tr>
<td></td>
<td>INCLUDE_LAPACKE</td>
</tr>
</tbody>
</table>
Table 7. **libFLAME Config Options**

<table>
<thead>
<tr>
<th>Feature</th>
<th>CMake Parameter(s)</th>
</tr>
</thead>
</table>
| Flags disabled by default                    | ENABLE_AUTODETECT_F77_UNDERSCORING
|                                               | ENABLE_BLAS3_FRNTEND_CNTL_TREES
|                                               | ENABLE_BUILTIN_BLAS
|                                               | ENABLE_CBLAS_INTERFACES
|                                               | ENABLE_DEFAULT_BLKSZ
|                                               | ENABLE_EXT_LAPACK_SUBPROBLEMS
|                                               | ENABLE_GOTO_INTERFACES
|                                               | ENABLE_GPU
|                                               | ENABLE_LIDM_ALIGNMENT
|                                               | ENABLE_MEMLK_CNTR
|                                               | ENABLE_MEMORY_ALIGNMENT2
|                                               | ENABLE_SUPER_MATRIX
|                                               | ENABLE_UPPERCASE_BLAS
|                                               | ENABLE_UPPERCASE_LAPACK
|                                               | ENABLE_WRAPPER
|                                               | ENABLE_XBLAS
| Enable uppercase APIs                         | ENABLE_UPPERCASE=ON                                                                |
| 32-bit integer size                          | ENABLE_ILP64=OFF                                                                   |
| 64-bit integer size                          | ENABLE_ILP64=ON                                                                   |
| AOCL-BLIS library path name                  | CMAKE_EXT_BLIS_LIBRARY_DEPENDENCY_PATH=<path to BLIS library>                     |
| AOCL-BLIS library name                       | EXT_BLIS_LIBNAME=BLIS library name                                                |
| Enable invoking ‘void’ return based          | ENABLE_F2C_DOTC=ON                                                                 |
| interface for BLAS functions DOTC and DOTU   |                                                                                   |
| Enable ‘void’ return type for libFLAME        | ENABLE_VOID_RETURN_COMPLEX_FUNCTION=ON                                             |
| functions such as cladiv/zladiv              |                                                                                   |

5. Provide the path to the BLIS library. It will be used in the linking stage while building the test suite.
6. To generate the Microsoft Visual Studio project in the out folder, click on the Generate button as shown in the following figure:

![Figure 8. libFLAME CMake Configurations](image)

5.3.1.2 Building the Project in Visual Studio GUI

Complete the following steps in the Microsoft Visual Studio GUI:

1. Open the project generated by CMake (build folder) in "Preparing Project with CMake GUI" on page 48.
2. To generate libFLAME binaries, build the AOCL-LibFLAME-Win project.
   
   The library files will generate in the bin folder based on the project settings.
   
   For example, *libflame/bin/Release/AOCL-LibFLAME-Win.dll* or *AOCL-LibFLAME-Win.lib*

5.3.2 Building libFLAME using Command-line Arguments

The project configuration and build procedures can be triggered from the command prompt as well. The corresponding steps are described in the following sections.
5.3.2.1 Configuring the Project in Command Prompt

In the libFLAME project folder, create a folder `out`. Open the command prompt in this directory and run the following command to configure the project:

```cmake
cmake -S .. -B . -G "Visual Studio 16 2019" -DCMAKE_BUILD_TYPE=Release
-DCMAKE_INSTALL_PREFIX="/path/to/install" -DCMAKE_INSTALL_BIN_DIR="/path/to/bin" -DCMAKE_INSTALL_LIB_DIR="/path/to/lib"
-DCMAKE_EXE_LINKER_FLAGS="-Wl,--start-group -Wl,--undefined-at-top -Wl,--end-group"
-DCMAKE_BUILD_TYPE="Release"
-DCMAKE_CXX_STANDARD="17"
-DCMAKE_CXX_FLAGS="-std=c++17 -fPIC" -DCMAKE_EXE_LINKER_FLAGS="-static -shared"
-DCMAKE_INSTALL_PREFIX="/path/to/install" -DCMAKE_INSTALL_BIN_DIR="/path/to/bin" -DCMAKE_INSTALL_LIB_DIR="/path/to/lib"
-DCMAKE_EXE_LINKER_FLAGS="-Wl,--start-group -Wl,--undefined-at-top -Wl,--end-group"
```

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

5.3.2.2 Building the Project in Command Prompt

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

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

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

5.3.3 Building and Running Test Suite

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

5.3.3.1 Running Test Suite

Copy the input files `input.global.general` and `input.global.operations` for the tests from `libflame\test` to the release folder. Run the tests from the command prompt as follows:

```cmake
Release> test_libFLAME.exe
```
Chapter 6 FFTW Library for AMD

AMD optimized version of Fast Fourier Transform Algorithm (FFTW) is a comprehensive collection of fast C routines for computing the Discrete Fourier Transform (DFT) and various special cases thereof that are optimized for AMD EPYC™ and other AMD “Zen”-based processors. It is an open-source implementation of FFTW. It can compute transforms of real and complex valued arrays of arbitrary size and dimension.

6.1 Installing

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

6.1.1 Building FFTW from Source on Linux

Complete the following steps to build AMD Optimized FFTW for AMD EPYC™ processor based on the architecture generation:


2. Depending on the target system and build environment, you must enable/disable the appropriate configure options. Set PATH and LD_LIBRARY_PATH to the MPI installation. In the case of building for AMD Optimized FFTW library with AOCC compiler, you must compile and setup OpenMPI with AOCC compiler.

Complete the following steps to compile it for EPYC™ processors:
Note: For a complete list of options and their description, type `./configure --help`.

- With GCC (default)

  **Double Precision FFTW libraries**

  ```
  $ ./configure --enable-sse2 --enable-avx --enable-avx2 --enable-mpi --enable-openmp --enable-shared --enable-amd-opt --enable-amd-mpifft --prefix=<your-install-dir>
  ```

  **Single Precision FFTW libraries**

  ```
  $ ./configure --enable-sse2 --enable-avx --enable-avx2 --enable-mpi --enable-openmp --enable-shared --enable-single --enable-amd-opt --enable-amd-mpifft --prefix=<your-install-dir>
  ```

  **Long double FFTW libraries**

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

  **Quad Precision FFTW libraries**

  ```
  $ ./configure --enable-shared --enable-openmp --enable-quad-precision --enable-amd-opt --prefix=<your-install-dir>
  ```

- With AOCC

  **Double Precision FFTW libraries**

  ```
  $ ./configure --enable-sse2 --enable-avx --enable-avx2 --enable-mpi --enable-openmp --enable-shared --enable-amd-opt --enable-amd-mpifft --prefix=<your-install-dir> CC=clang F77=flang FC=flang
  ```

  **Single Precision FFTW libraries**

  ```
  $ ./configure --enable-sse2 --enable-avx --enable-avx2 --enable-mpi --enable-openmp --enable-shared --enable-single --enable-amd-opt --enable-amd-mpifft --prefix=<your-install-dir> CC=clang F77=flang FC=flang
  ```

  **Long double FFTW libraries**

  ```
  $ ./configure --enable-shared --enable-openmp --enable-mpi --enable-long-double --enable-amd-opt --enable-amd-mpifft --prefix=<your-install-dir> CC=clang F77=flang FC=flang
  ```

  **Quad FFTW libraries**

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

AMD optimized fast planner is added as an extension to the original planner to improve the planning time of various planning modes in general and PATIENT mode in particular.

The configure user option `--enable-amd-fast-planner` when given in addition to `--enable-amd-opt` enables this new fast planner.
An optional configure option AMD_ARCH is supported, that can be set to the CPU architecture values, such as auto, znver1, znver2, or znver3 for AMD EPYC™ and other AMD “Zen”-based processors.

Additional config and build options to enable specific optimizations are covered in the section “AMD Optimized FFTW Tuning Guidelines” on page 100.

The configure option --enable-amd-opt is the mandatory master optimization switch that must be set for enabling any other optional configure options, such as:

- --enable-amd-mpifft
- --enable-amd-mpivader-limit
- --enable-amd-trans
- --enable-amd-fast-planner
- --enable-amd-top-n-planner
- --enable-amd-app-opt

3. Build the library:

   $ make

4. Install the library in the preferred path:

   $ make install

5. Verify the installed library:

   $ make check

### 6.1.2 Building FFTW from Source on Windows

FFTW uses CMake along with Microsoft Visual Studio for building binaries from the sources on Windows. This section explains the GUI and command-line schemes for building the binaries and test suite.

**Prerequisites**

The following prerequisites must be met:

- Windows Server 2019
- A suitable MPI library installation along with the appropriate environment variables on the host machine
- LLVM 13 for AMD “Zen3” support (or LLVM 11 for AMD “Zen2” support)
- LLVM plug-in for Microsoft Visual Studio (if latest version of LLVM is installed separately, this plugin enables linking Visual Studio with the installed LLVM tool-chain)
- CMake 3.0 or later
- MPI compiler
- Microsoft Visual Studio 2019 build 16.8.7
• Microsoft Visual Studio tools
  – Python development
  – Desktop development with C++: C++ Clang-Cl for v142 build tool (x64 or x86)

6.1.2.1 Using CMake GUI to Build

Complete the following steps in the CMake GUI:

1. Set the source (folder containing FFTW source code) and build (folder in which the project files will be generated, for example, out) folder paths.

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

3. Set the generator to Visual Studio 16 2019 and the compiler to ClangCl or LLVM.

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

<table>
<thead>
<tr>
<th>Feature</th>
<th>CMake Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Build type (Release or Debug mode)</td>
<td>CMAKE_BUILD_TYPE=Release/Debug</td>
</tr>
<tr>
<td>AMD CPU architecture (AMD “Zen”/AMD “Zen2”/AMD “Zen3”)</td>
<td>AMD_ARCH: STRING=znver1/znver2/znver3</td>
</tr>
<tr>
<td>Shared library without multithreading</td>
<td>BUILD_SHARED_LIBS=ON</td>
</tr>
<tr>
<td></td>
<td>ENABLE_OPENMP=OFF</td>
</tr>
<tr>
<td></td>
<td>ENABLE_THREADS=OFF</td>
</tr>
<tr>
<td>Shared library with multithreading</td>
<td>BUILD_SHARED_LIBS=ON</td>
</tr>
<tr>
<td></td>
<td>ENABLE_OPENMP=ON</td>
</tr>
<tr>
<td>Static library without multithreading</td>
<td>BUILD_SHARED_LIBS=OFF</td>
</tr>
<tr>
<td></td>
<td>ENABLE_OPENMP=OFF</td>
</tr>
<tr>
<td>Static library with multithreading</td>
<td>BUILD_SHARED_LIBS=OFF</td>
</tr>
<tr>
<td></td>
<td>ENABLE_OPENMP=ON</td>
</tr>
<tr>
<td>Use Threads instead of OpenMP for multithreading</td>
<td>ENABLE_THREADS=ON</td>
</tr>
<tr>
<td></td>
<td>WITH_COMBINED_THREADS=ON</td>
</tr>
<tr>
<td>Use both Threads and OpenMP for multithreading</td>
<td>ENABLE_THREADS=ON</td>
</tr>
<tr>
<td></td>
<td>ENABLE_OPENMP=ON</td>
</tr>
<tr>
<td>Flags for enhanced instruction set support</td>
<td>ENABLE_SSE=ON</td>
</tr>
<tr>
<td></td>
<td>ENABLE_SSE2=ON</td>
</tr>
<tr>
<td></td>
<td>ENABLE_AVX=ON</td>
</tr>
<tr>
<td></td>
<td>ENABLE_AVX2=ON</td>
</tr>
<tr>
<td>Flags for single and long double</td>
<td>ENABLE_FLOAT=ON</td>
</tr>
<tr>
<td></td>
<td>ENABLE_LONG_DOUBLE=ON</td>
</tr>
<tr>
<td>Build tests directory and generate test applications</td>
<td>BUILD_TESTS=ON</td>
</tr>
</tbody>
</table>
Table 8. FFTW Config Options

<table>
<thead>
<tr>
<th>Feature</th>
<th>CMake Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enables MPI lib</td>
<td>ENABLE_MPI=ON</td>
</tr>
<tr>
<td>Enables AMD optimizations</td>
<td>ENABLE_AMD_OPT=ON</td>
</tr>
<tr>
<td>Enables AMD MPI FFT optimizations</td>
<td>ENABLE_AMD_MPIFFT=ON</td>
</tr>
<tr>
<td>Enables AMD optimized transpose</td>
<td>ENABLE_AMD_TRANS=ON</td>
</tr>
<tr>
<td>Enables AMD optimizations for HPC/Scientific applications</td>
<td>ENABLE_AMD_APP_OPT=ON</td>
</tr>
</tbody>
</table>

Note: ENABLE_QUAD_PRECISION and wisdom executable are currently not supported on Windows.

5. Select the available and recommended options as follows:

![Figure 9. FFTW CMake Config Options](image)

6. Click the Generate button and then Open Project.
6.1.2.2 Using Command-line Arguments to Build

Complete the following steps to trigger the project configuration and build procedures from the command prompt:

1. In the FFTW project folder, create a folder out. Open the command prompt in this directory and run the following command to configure the project:

   ```
   cmake .. -DBUILD_TESTS=ON -D[other options1] -D[other options2] -T ClangCl -G "Visual Studio 16 2019" & & cmake --build . --config Release
   ```

2. Refer Table 8 and update the parameter options in the command according to the project requirements.

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

3. To verify the installed library, copy the test scripts from \win\tests to \out\Release and run python fftw_check.py.

6.1.3 Using Pre-built Libraries

The AMD optimized FFTW library binaries for Linux are available in the following URL:

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

The AMD optimized FFTW binary for Linux and Windows can also be installed from the AOCL master installer (tar packages for Linux and zip packages for Windows) available in the following URL:

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

The tar and zip files include pre-built binaries of other AMD libraries BLIS, libFLAME, ScaLAPACK, LibM, aocl-sparse, RNG, and AMD Secure RNG.

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

6.2 Usage

Sample programs and executable binaries demonstrating the usage of FFTW APIs and performance benchmarking are available in tests/ and mpi/ directories for Linux and out/Release directory for Windows.

6.2.1 Sample Programs for Single-threaded and Multi-threaded FFTW

To run single-threaded test, execute the following command:

```bash
$ bench -opatient -s [i|o][r][c][f|b]<size>
```

Where,
• i/o means in-place or out-of-place. Out of place is the default.
• r/c means real or complex transform. Complex is the default.
• f/b means forward or backward transform. Forward is the default.
• <size> is an arbitrary multidimensional sequence of integers separated by the character 'x'.

Check the tuning guidelines for single-threaded test execution in “AMD Optimized FFTW Tuning Guidelines” on page 100.

To run multi-threaded test, execute the following command:

```bash
$bench -opatient -onthreads=N -s [i|o][r|c][f|b]<size>
```

Where, N is number of threads.

Check the tuning guidelines for multi-threaded test execution in the section “AMD Optimized FFTW Tuning Guidelines” on page 100.

### 6.2.2 Sample Programs for MPI FFTW

```bash
$mpirun -np N mpi-bench -opatient -s [i|o][r|c][f|b]<size>
```

Where, N is the number of processes.

Check the tuning guidelines for MPI test execution in the section “AMD Optimized FFTW Tuning Guidelines” on page 100.

### 6.2.3 Additional Options

• `-owisdom`
  
  On startup, read wisdom from the file `wis.dat` in the current directory (if it exists).
  
  On completion, write accumulated wisdom to `wis.dat` (overwriting if file exists).
  
  This bypasses the planner next time onwards and directly executes the read plan from wisdom.

• `--verify <problem>`
  
  Verify that the FFTW is computing correctly. It does not output anything unless there is an error.

• `-v<n>`
  
  Set verbosity to `<n>` or 1 if `<n>` is omitted. -v2 will output the created plans.

**Notes:**

1. The names of windows FFTW test bench application has .exe extension (`bench.exe` and `mpi-bench.exe`).
2. The folder /win/tests/ includes Windows benchmark scripts for single-threaded, multi-threaded and MPI FFT execution for standard sizes. A README file is also provided with the instructions to run these benchmark scripts.

To display the AOCL version number of FFTW library, application must call the following FFTW API `fftw_aoclversion()`.

The test bench executables of FFTW support the display of AOCL version using the `--info-all` option.
Chapter 7  AMD LibM

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

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

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

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

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

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

Where,

• v – vector
• r – real
• a – array
• <type> - ‘s’ for single precision and ‘d’ for double precision
• <vec_size> - 2 or 4 for 2 or 4 element vector respectively
• <func> - function name, such as ‘exp’ and ‘expf’

For example, a single precision 4 element version of exp has the signature:

```
__m128 vrs4_expf(__m128 x)
```
The list of available vector functions is as follows:

**Note:** All the functions have an 'amd_' prefix and it is omitted from the list to reduce the length.

- **Exponential**
  - vrs8_expf and vrs8_exp2f
  - vrs4_expf, vrs4_exp2f, vrs4_exp10f, and vrs4_expm1f
  - vrsa_expf, vrsa_exp2f, vrsa_exp10f, and vrsa_expm1f
  - vrd2_exp, vrd2_exp2, vrd2_exp10, vrd2_expm1, vrd4_exp, and vrd4_exp2
  - vrda_exp, vrda_exp2, vrda_exp10, and vrda_expm1

- **Logarithmic**
  - vrs8_logf, vrs8_log2f, and vrs8_log10f
  - vrs4_logf, vrs4_log2f, vrs4_log10f, and vrs4_log1pf
  - vrd4_log and vrd4_log2
  - vrsa_logf, vrsa_log2f, vrsa_log10f, and vrsa_log1pf
  - vrd2_log, vrd2_log2, vrd2_log10, and vrd2_log1p
  - vrda_log, vrda_log2, vrda_log10, vrda_log1p

- **Trigonometric**
  - vrs4_cosf, vrs8_cosf, vrs4_sinf, and vrs8_sinf
  - vrsa_cosf, vrsa_sinf, and vrsa_sincosf
  - vrd4_sin, vrd4_cos, and vrd4_tan
  - vrd2_cos, vrd2_sin, vrd2_tan, and vrd2_sincos
  - vrda_cos, vrda_sin, and vrda_sincos

- **Inverse Trigonometric**
  - vrs4_acosf, vrs4_asinf, and vrs8_asinf
  - vrs4_atanf, vrs8_atanf, vrd2_atan

- **Hyperbolic**
  - vrs4_coshf and vrs4_tanhf
  - vrs8_coshf and vrs8_tanhf

- **Power**
  - vrs4_powf, vrd2_pow, vrd4_pow, vrs8_powf, and vrsa_powf

The following scalar functions are present in the library:

They can be called by a standard C99 function call and naming convention and must be linked with AMD LibM before standard †1ibm.
For example:

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

The following functions have vector variants in AMD LibM:

- **Trigonometric**
  - cosf, cos, sinf, sin, tanf, tan, sincosf, and sincos
- **Inverse Trigonometric**
  - acosf, acos, asinf, asin, atanf, atan, atan2f, and atan2
- **Hyperbolic**
  - coshf, cosh, sinh, sinh, tanhf, and tanh
- **Inverse Hyperbolic**
  - acoshf, acosh, asinhf, asinh, atanhf, and atanh
- **Exponential and Logarithmic**
  - expf, exp, exp2f, exp2, exp10f, exp10, expm1f, and expm1
  - logf, log, log10f, log10, log2f, log2, log1pf, and log1p
  - logbf, logb, ilogbf, and ilogb
  - modff, modf, frexpf, frexp, ldexpf, and ldexp
  - scalbnf, scalbn, scalblnf, and scalbln
- **Power and Absolute Value**
  - powf, pow, fastpow, cbutf, cbt, sqrtf, sqrt, hypotf, and hypot
  - fabsf and fabs
- **Nearest integer**
  - ceilf, ceil, floorf, floor, truncf, and trunc
  - rintf, rint, roundf, round, nearbyintf, and nearbyint
  - lrintf, lrint, llrintf, and llrint
  - lroundf, lround, llroundf, and llround
- **Remainder**
  - fmodf, fmod, remainderf, and remainder
• Manipulation
  – copysignf, copysign, nanf, nan, finitex, and finite
  – nextafterf, nextafter, nexttowardf, and nexttoward
• Maximum, Minimum, and Difference
  fdimf, fdim, fmaxf, fmax, fminf, and fmin

7.1 Installation on Linux

AMD LibM binary for Linux can be found in the following URL:

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

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

7.2 Compiling AMD LibM

Minimum software requirements for compilation:
• GCC 9.2 or later
• Clang 9.0 or later
• Glibc 2.29 or later
• Virtualenv with python3
• SCons 3.0.5

Complete the following steps to compile AMD LibM:
2. Navigate to the LibM folder and checkout to the branch aocl-3.1:
   cd aocl-libm-ose
git checkout aocl-3.1
3. Create a virtual environment:
   virtualenv -p python3 .venv3
4. Activate the virtual environment:
   source .venv3/bin/activate
5. Install SCons:
   pip install scons
6. Compile AMD LibM:

```
  $ scons -j32
```

Additional parameters:
- `install --prefix=<path to install>`
- `ALM_CC=<gcc/clang exe path>`
- `Verbosity options: --verbose=1`
- `Debug mode build: --debug_mode=libs`

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

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

### 7.3 Usage

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

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

   The Linux libraries may have a dependency on the system math library. When linking AMD LibM, ensure that it precedes the system math library in the link order that is, `-lalm` must appear before `-lm`.
   
   The explicit linking of the system math library is required when using the GCC/AOCC compiler.
   
   Such explicit linking is not required with the g++ compiler (for C++).

   Example: `myprogram.c`

   ```c
   #include <stdio.h>
   #include <math.h>

   int main() {
     float f = 3.14f;
     printf ("%f\n", expf(f));
     return 0;
   }
   ```

   To use AMD LibM scalar functions, use the following commands:

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

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

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

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

   ```c
   #include <stdio.h>
   #include <math.h>

   int main() {
     float f = 3.14f;
     printf ("%f\n", expf(f));
     return 0;
   }
   ```

   ```sh
   $ export LD_LIBRARY_PATH=<Path to libalm.so>:${LD_LIBRARY_PATH};
   $ cc -Wl,-std=c99 myprogram.c -o myprogram -L<Path to libalm.so> -lalm -lm; (cc can be ‘gcc’ or ‘clang’).
   $ ./myprogram;
   ```
### Example: myprogram.c

```c
#include "amdlibm_vec.h"
__m128 vrs4_expf (__m128 x);

__m128
test_expf_v4s(float *ip, float *out)
{
    __m128 ip4 = _mm_set_ps(ip1[3], ip1[2], ip1[1], ip1[0]);
    __m128 op4 = vrs4_expf(ip4);
    _mm_store_ps(&out[0], op4);
    return op4;
}
```

You can compile myprogram.c as follows:

```
$ export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/path/to/AMD LibM
$ clang -Wall -std=c99 -ffastmath myprogram.c -o myprogram -L<path to libalm.so> -lalm -lm
```

## 7.4 Building AMD LibM on Windows

Minimum software requirements for compilation:
- Windows 10 or Windows Server 2019
- LLVM compiler V13.0 (or LLVM compiler V11.0 for AMD “Zen2” support)
- Microsoft Visual Studio 2019 build 16
- Windows SDK Version 10.0.18362.0
- Virtualenv with python3
- SCons 4.2.0

Complete the following steps to compile AMD LibM on Windows:

1. Download source from GitHub ([https://github.com/amd/aocl-libm-ose](https://github.com/amd/aocl-libm-ose)).
2. Navigate to the folder:
   ```
   cd aocl-libm-ose
   ```
3. Install virtualenv:
   ```
   pip install virtualenv
   ```
4. Initialize the environment for correct architecture using Visual Studio vcvarsall.bat file using following command:
   ```
   "C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\VC\Auxiliary\Build\vcvarsall.bat" amd64
   ```
5. Activate virtual environment and install SCons inside:

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

6. Build the project using clang compiler:

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

For example:

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

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

```
aocl-libm-ose/build/aocl-release/src/
```
Chapter 8  ScaLAPACK Library for AMD

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

8.1 Installation

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

8.1.1 Building ScaLAPACK from Source

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

GitHub URL: [https://github.com/amd/aocl-scalapack](https://github.com/amd/aocl-scalapack)

**Prerequisites**

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

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

Complete the following steps to build ScaLAPACK from source:

1. Clone the GitHub repository ([https://github.com/amd/aocl-scalapack.git](https://github.com/amd/aocl-scalapack.git)).

2. Execute the command:

   ```
   $ cd scalapack
   ```

3. CMake as follows:

   a. Create a new directory. For example, build:
      ```
      $ mkdir build
      $ cd build
      ```

   b. Set PATH and LD_LIBRARY_PATH appropriately to the MPI installation.
c. Run `cmake` command based on the compiler and the type of library generation required.

**Table 9. Compiler and Type of Library**

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Library Type</th>
<th>Threading</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC</td>
<td>Static</td>
<td>Single-thread</td>
<td>$ cmake .. -DBUILD_SHARED_LIBS=OFF -DBLAS_LIBRARIES=&quot;&lt;path to BLIS library&gt;/libblis.a&quot; -DLAPACK_LIBRARIES=&quot;&lt;path to libflame library&gt;/libflame.a&quot; -DCMAKE_C_COMPILER=mpicc -DCMAKE_Fortran_COMPILER=mpif90 -DUSE_OPTIMIZED_LAPACK_BLAS=OFF [-D DENABLE_ILP64=ON]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Multi-thread</td>
<td>$ cmake .. -DBUILD_SHARED_LIBS=OFF -DBLAS_LIBRARIES=&quot;-fopenmp &lt;path to BLIS library&gt;/libblis-mt.a&quot; -DLAPACK_LIBRARIES=&quot;&lt;path to libflame library&gt;/libflame.a&quot; -DCMAKE_C_COMPILER=mpicc -DCMAKE_Fortran_COMPILER=mpif90 -DUSE_OPTIMIZED_LAPACK_BLAS=OFF [-D DENABLE_ILP64=ON]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Shared</td>
<td>$ cmake .. -DBUILD_SHARED_LIBS=ON -DBLAS_LIBRARIES=&quot;&lt;path to BLIS library&gt;/libblis.so&quot; -DLAPACK_LIBRARIES=&quot;&lt;path to libflame library&gt;/libflame.so&quot; -DCMAKE_C_COMPILER=mpicc -DCMAKE_Fortran_COMPILER=mpif90 -DUSE_OPTIMIZED_LAPACK_BLAS=OFF [-D DENABLE_ILP64=ON]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Multi-thread</td>
<td>$ cmake .. -DBUILD_SHARED_LIBS=ON -DBLAS_LIBRARIES=&quot;-fopenmp &lt;path to BLIS library&gt;/libblis-mt.so&quot; -DLAPACK_LIBRARIES=&quot;&lt;path to libflame library&gt;/libflame.so&quot; -DCMAKE_C_COMPILER=mpicc -DCMAKE_Fortran_COMPILER=mpif90 -DUSE_OPTIMIZED_LAPACK_BLAS=OFF [-D DENABLE_ILP64=ON]</td>
</tr>
</tbody>
</table>
d. Ensure CMake locates libFLAME and BLIS libraries. On completion, a message, “**LAPACK routine dgesv is found: 1**” similar to the following in CMake output is displayed:

```
...
-- CHECKING BLAS AND LAPACK LIBRARIES
-- --> LAPACK supplied by user is <path>/libflame.a.
-- --> LAPACK routine dgesv is found: 1.
-- --> LAPACK supplied by user is WORKING, will use <path>/libflame.a.
-- BLAS library: <path>/libblis.a
-- LAPACK library: <path>/libflame.a
...
```

e. Compile the code:

```
$ make -j
```

When the building process is complete, the ScaLAPACK library is created in the lib directory. The test application binaries are generated in the `<aocl-scalapack>/build/TESTING` folder.
8.1.2 Using Pre-built Libraries

AMD optimized ScaLAPACK library binaries for Linux are available in the following URLs:

- [https://github.com/amd/aocl-scalapack/releases](https://github.com/amd/aocl-scalapack/releases)

Also, AMD optimized ScaLAPACK binary can be installed from the AOCL master installer tar file available in the following URL:


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

8.2 Usage

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

```
$ cd scalapack/TESTING
```

8.3 Building ScaLAPACK from Source on Windows

GitHub URL: [https://github.com/amd/aocl-scalapack](https://github.com/amd/aocl-scalapack)

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

**Prerequisites**

The following prerequisites must be met:

- Windows Server 2019
- LLVM 13 for AMD “Zen3” support (or LLVM 11 for AMD “Zen2” support)
- LLVM plug-in for Microsoft Visual Studio (if latest version of LLVM is installed separately, this plug-in enables linking Microsoft Visual Studio with the installed LLVM tool-chain)
- CMake 3.0 or later
- MPI compiler
- Fortran 90 compiler
- Microsoft Visual Studio 2019 build 16.8.7
- Microsoft Visual Studio tools
  - Python development
  - Desktop development with C++: C++ Clang-Cl for v142 build tool (x64 or x86)
8.3.1 Building ScaLAPACK Using GUI

8.3.1.1 Preparing Project with CMake GUI

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

1. Set the source (folder containing aocl-scalapack source code) and build (folder in which the project files will be generated, for example, out) folder paths. It is not recommended to use the folder named build as it is already used for Linux build system.

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

3. Set the generator to Visual Studio 16 2019 and the compiler to ClangCl or LLVM.

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

Table 10. ScaLAPACK CMake Parameter List

<table>
<thead>
<tr>
<th>Build Feature</th>
<th>CMake Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select debug or Release mode build</td>
<td>CMAKE_BUILD_TYPE=Debug/Release</td>
</tr>
<tr>
<td>Shared library</td>
<td>BUILD_SHARED_LIBS=ON BUILD_STATIC_LIBS=OFF</td>
</tr>
<tr>
<td>Static library</td>
<td>BUILD_STATIC_LIBS=ON BUILD_SHARED_LIBS=OFF</td>
</tr>
<tr>
<td>Provide external BLAS/BLIS library</td>
<td>BLAS_LIBRARIES=&lt;Path to BLAS/BLIS lib&gt;</td>
</tr>
<tr>
<td>Provide external lapack/libflame library</td>
<td>LAPACK_LIBRARIES=&lt;Path to lapack/libflame lib&gt;</td>
</tr>
<tr>
<td>Use fortran-to-C converted files</td>
<td>USE_F2C=ON</td>
</tr>
<tr>
<td>Integer bit length:</td>
<td></td>
</tr>
<tr>
<td>• ON =&gt; 64-bit integer length</td>
<td>ENABLE_ILP64</td>
</tr>
<tr>
<td>• OFF =&gt; 32-bit integer length</td>
<td></td>
</tr>
<tr>
<td>Flags disabled by default</td>
<td>USE_OPTIMIZED_LAPACK_BLAS</td>
</tr>
</tbody>
</table>
5. Select the available and recommended options as follows:

![Figure 10. ScaLAPACK CMake Options](image)

Figure 10. ScaLAPACK CMake Options

![Figure 11. ScaLAPACK CMake Config Options](image)

Figure 11. ScaLAPACK CMake Config Options

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

### 8.3.1.2 Building the Project in Visual Studio GUI

Complete the following steps in Microsoft Visual Studio GUI:

1. Open the project generated by CMake (build folder) in “Preparing Project with CMake GUI” on page 71.

2. To generate the ScaLAPACK binaries, build the **ScaLAPACK** project. The library files would be generated in the folder **out** based on the project settings.

For example:
aocl-scalapack/out/lib/Release/scalapack.lib
aocl-scalapack/out/Testing/Release/scalapack.dll

8.3.2 Building ScaLAPACK using Command-line Arguments

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

8.3.2.1 Configuring the Project in Command Prompt

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

1. In the ScaLAPACK project folder, create a folder `out`.
2. Open the command prompt in that directory and run the following command:

   ```
   cmake -S .. -B . -G "Visual Studio 16 2019" -DCMAKE_BUILD_TYPE=Release
   -DBUILD_SHARED_LIBS=ON
   -DBUILD_STATIC_LIBS=OFF -DUSE_F2C=ON -DBLAS_LIBRARIES="<path to BLIS library>/AOCL-LibBlis-Win-MT-dll.lib"
   -DLAPACK_LIBRARIES="<path to libflame library>/AOCL-LibFLAME-Win-MT-dll.lib"
   Refer Table 10 to update the parameter options in the command according to the project requirements.
   ```

8.3.2.2 Building the Project in Command Prompt

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

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

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

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

8.3.2.3 Building and Running the Individual Tests

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

8.3.2.4 Running Individual Tests

The test application binaries are generated in the folder `<aocl-scalapack>/out/Testing/Release` or `<aocl-scalapack>/out/Testing/Debug` based on the project settings. Run the tests from the command prompt as follows:

   ```
   Release> mpiexec xcbrd.exe
   ```
Chapter 9  AMD Random Number Generator

The AMD Random Number Generator (RNG) library is a pseudo-random number generator library. It provides a comprehensive set of statistical distribution functions and various uniform distribution generators (base generators) including Wichmann-Hill and Mersenne Twister. The library contains five base generators and twenty-three distribution generators. In addition, you can supply a custom-built generator as the base generator for all the distribution generators.

9.1  Installation

The AMD RNG binary for Linux is available in the following URL:


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

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

The tar file includes pre-built binaries of other AMD Libraries BLIS, libFLAME, FFTW, LibM, ScaLAPACK, aocl-sparse, and AMD Secure RNG. Following are the supported compilers to build RNG from source:

- GCC 11.1.0
- AOCC 3.1 and 3.2

9.2  Usage

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

The following is a sample Makefile for an application that uses the AMD RNG 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
```

For more information, refer the examples directory in the AMD RNG library install location.
Chapter 10  AMD Secure RNG

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

10.1 Installation

The AMD Secure RNG library can be downloaded from following URL:


Also, AMD Secure RNG can be installed from the AOCL master installer tar file available in the following URL:

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

The tar file includes pre-built binaries of other AMD Libraries BLIS, libFLAME, LibM, ScaLAPACK, FFTW, aocl-sparse, and AMD RNG library. Following are the supported compilers to build RNG from source:

- GCC 11.1.0
- AOCC 3.1 and 3.2

10.2 Usage

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

- `include/secrng.h` — A header file that has declaration of all the library APIs.
- `src_lib/secrng.c` — Contains the implementation of the APIs.
- `src_test/secrng_test.c` — Test application to test all the library APIs.
- `Makefile` — To compile the library and test the application.

You can use the included `makefile` to compile the source files and generate dynamic and static libraries. Then, you can link it to your application and invoke the required APIs.
The following code snippet shows a sample usage of the library API:

```c
//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", rng64);
    else
        printf("Failure in retrieving random value using RDRAND!\n");

    //Get a range of 64-bit random values
    uint64_t* rng64_arr = (uint64_t*) malloc(sizeof(uint64_t) * N);
    ret = get_rdrand64u_arr(rng64_arr, N, 0);

    if (ret == SECRNG_SUCCESS)
    {
        printf("RDRAND for %u 64-bit random values succeeded!\n", N);
        printf("First 10 values in the range : \n");
        for (int i = 0; i < (N > 10? 10 : N); i++)
            printf("%lu\n", rng64_arr[i]);
    }
    else
        printf("Failure in retrieving array of random values using RDRAND!\n");
}
else
{
    printf("No support for RDRAND!\n");
}
```

In the example, `get_rdrand64u` is invoked to return a single 64-bit random value and `get_rdrand64u_arr` is used to return an array of 1000 64-bit random values.
Chapter 11  AOCL-Sparse

AOCL-Sparse is a library containing basic linear algebra subroutines for the sparse matrices and vectors optimized for AMD EPYC™ and other AMD “Zen”-based 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 3 functions describe the operations between a matrix in sparse format and a matrix in dense/sparse format.
- Sparse Level 2 functions describe the operations between a matrix in sparse format and a vector in dense format.
- Sparse Format Conversion functions describe operations on a matrix in sparse format to obtain a different matrix format.

The list of supported functions is as follows:

- Sparse Level 3
  - aoclsparse_xcsrm (Single and double precision)

- Sparse Level 2
  - aoclsparse_xellmv (Single and double precision)
  - aoclsparse_xellmv (Single and double precision)
  - aoclsparse_xdiamv (Single and double precision)
  - aoclsparse_xbsrmv (Single and double precision)
  - aoclsparse_xcsrsv (Single and double precision)

- Sparse Auxiliary
  - aoclsparse_get_version
  - aoclsparse_create_mat_descr
  - aoclsparse_destroy_mat_descr
  - aoclsparse_copy_mat_descr
  - aoclsparse_set_mat_fill_mode
  - aoclsparse_get_mat_fill_mode
  - aoclsparse_set_mat_diag_type
  - aoclsparse_get_mat_diag_type
  - aoclsparse_create_mat_csr
  - aoclsparse_destroy_mat_csr
• Conversion
  – aoclsparse_csr2ell_width
  – aoclsparse_xcsr2ell (Single and double precision)
  – aoclsparse_csr2dia_ndiag
  – aoclsparse_xcsr2dia (Single and double precision)
  – aoclsparse_csr2bsr_nnz
  – aoclsparse_xcsr2bsr (Single and double precision)
  – aoclsparse_xcsr2csc (Single and double precision)
  – aoclsparse_xcsr2dense (Single and double precision)


11.1 Installation

11.1.1 Building AOCL-Sparse from Source

The following compile-time dependencies must be met:

• Git
• CMake 3.5 or later

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

2. Clone the Git repository (https://github.com/amd/aocl-sparse.git).
3. Run the command:
   ```
   cd aocl-sparse
   ```
4. Create the build directory and change to it:
   ```
   $ mkdir -p build/release
cd build/release
   ```
5. Run CMake as per the required compiler and library type

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Library Type</th>
<th>ILP 64 Support</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>G++ (Default)</td>
<td>Static</td>
<td>OFF (Default)</td>
<td>cmake ../.. -DBUILD_SHARED_LIBS=OFF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ON</td>
<td>cmake ../.. -DBUILD_SHARED_LIBS=OFF -DBUILD_ILP64=ON</td>
</tr>
<tr>
<td></td>
<td>Shared (Default)</td>
<td>OFF (Default)</td>
<td>cmake ../..</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ON</td>
<td>$ cmake ../.. -DBUILD_ILP64=ON</td>
</tr>
<tr>
<td>AOCC</td>
<td>Static</td>
<td>OFF (Default)</td>
<td>cmake ../.. -DCMAKE_CXX_COMPILER=clang++ -DBUILD_SHARED_LIBS=OFF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ON</td>
<td>cmake ../.. -DCMAKE_CXX_COMPILER=clang++ -DBUILD_SHARED_LIBS=OFF -DBUILD_ILP64=ON</td>
</tr>
<tr>
<td></td>
<td>Shared (Default)</td>
<td>OFF (Default)</td>
<td>cmake ../.. -DCMAKE_CXX_COMPILER=clang++</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ON</td>
<td>$ cmake ../.. -DCMAKE_CXX_COMPILER=clang++ -DBUILD_ILP64=ON</td>
</tr>
</tbody>
</table>

6. If required, use the following CMake options:
   - Use `-DCMAKE_INSTALL_PREFIX=<path>` to choose the custom path. The default install path is `/opt/aoclsparse/`.
   - Use `-DBUILD_CLIENTS_BENCHMARKS=ON` to build the test application along with the aocl-sparse library. This is OFF by default.

7. Compile the aocl-sparse library:
   ```
   $ make -j$(nproc)
   ```

8. Install aocl-sparse to the directory `/opt/aoclsparse` or a custom path:
   ```
   $ make install
   ```

### 11.1.2 Simple Test

After compiling the library with benchmarks, run the following aocl-sparse example to test the installation:
1. Navigate to the test binary directory:

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

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

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

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

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

### 11.1.3 Using Pre-built Libraries

You can find the AMD optimized aocl-sparse library binaries for Linux in the following URLs:

- [https://github.com/amd/aocl-sparse/releases](https://github.com/amd/aocl-sparse/releases)

Also, you can install aocl-sparse binary from the AOCL master installer tar file available in the following URL:


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

### 11.2 Usage

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

   $ cd <New Revision> aocl-sparse/tests/examples

#### 11.2.1 Use by Applications

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

**Example**

With Static Library

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

With Dynamic Library

```
g++ sample_csrmv.cpp -I<path-to-aocl-sparse-header> <path-to-aocl-sparse-library>/libaoclsparse.so -o test_aoclsparse.x
```
The following is a sample *cpp* file depicting the AOCL-Sparse dcsrmv API usage:

```cpp
//file :sample_csrmv.cpp
#include "aoclsparse.h"
#include <iostream>
#define M 5
#define N 5
#define NNZ 8
int main(int argc, char* argv[]) {
    aoclsparse_operation trans = aoclsparse_operation_none;
double alpha = 1.0;
double beta = 0.0;

    // Print aoclsparse version
    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] = {1.0, 6.0, 1.050e+01, 1.500e-02, 2.505e+02, -2.800e+02, 3.332e+01, 1.200e+01};
    // Initialise vectors
double x[N] = { 1.0, 2.0, 3.0, 4.0, 5.0};
double y[M];
std::cout << "Invoking aoclsparse_dcsrmv..");
    //Invoke SPMV API for CSR storage format(double precision)
aoclsparse_dcsrmv(trans,
                        &alpha,
                        M,
                        N,
                        NNZ,
                        csr_val,
                        csr_col_ind,
                        csr_row_ptr,
                        descr,
                        x,
                        &beta,
                        y);
    std::cout << "Done." << std::endl;
    std::cout << "Output Vector:" << std::endl;
    for(aoclsparse_int i=0;i < M; i++)
        std::cout << y[i] << std::endl;
aoclsparse_destroy_mat_descr(descr);
    return 0;
}
```
A sample compilation command with the gcc compiler for the above code:

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

### 11.3 Build AOCL-Sparse from Source on Windows

GitHub URL: [https://github.com/amd/aocl-sparse](https://github.com/amd/aocl-sparse)

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

**Prerequisites**

For more information, refer to the Prerequisites sub-section in section “Build BLIS from Source on Windows” on page 38.

#### 11.3.1 Building AOCL-Sparse Using GUI

**11.3.1.1 Preparing Project with CMake GUI**

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

1. Set the source (folder containing aocl-sparse source code) and build (folder in which the project files will be generated, for example, out) folder paths. It is not recommended to use the folder named build as it is already used for Linux build system.

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

3. Set the generator to Visual Studio 16 2019 and the compiler to ClangCl.

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

**Table 12. AOCL-Sparse CMake Parameter List**

<table>
<thead>
<tr>
<th>Feature</th>
<th>CMake Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD “Zen” architecture for building the projects (znver1, znver2, znver3, and native)</td>
<td>AMD_ARCH</td>
</tr>
<tr>
<td>Integer length:</td>
<td></td>
</tr>
<tr>
<td>• ON =&gt; 64-bit integer length</td>
<td>BUILD_ILP64</td>
</tr>
<tr>
<td>• OFF =&gt; 32-bit integer length</td>
<td></td>
</tr>
<tr>
<td>Build Static (OFF) or Dynamic/Shared (ON) library</td>
<td>BUILD_SHARED_LIBS</td>
</tr>
<tr>
<td>ON =&gt; Build client benchmarking</td>
<td>BUILD_CLIENTS_BENCHMARKS</td>
</tr>
</tbody>
</table>
5. Select the available and recommended options as follows:

![Figure 12. AOCL-Sparse CMake Config Options]

6. Click the Generate button and then Open Project.

11.3.1.2 Building the Project in Visual Studio GUI

Complete the following steps in Microsoft Visual Studio GUI:

1. Open the project generated by CMake (build folder) in “Preparing Project with CMake GUI” on page 82.

2. To generate the AOCL-Sparse binaries, build the AOCL-Sparse project. The library files would be generated in the folder bin based on the project settings.

For example:

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

aoclsparse.lib

11.3.2 Building AOCL-Sparse using Command-line Arguments

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

11.3.2.1 Configuring the Project in Command Prompt

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

1. In the AOCL-Sparse project folder, create a folder out.

2. Open the command prompt in that directory and run the following command:

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

Refer Table 12 to update the parameter options in the command according to the project requirements.
11.3.2.2 Building the Project in Command Prompt

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

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

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

11.3.2.3 Building and Running the Test Suite

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

11.3.2.4 Running Individual Tests

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

```
 aoclsparse-bench.exe --function=ellmv --precision=d --verify=1
```
Chapter 12  AOCL Spack Recipes

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

Notes:

1. From AOCL 2.2 release onwards, the Spack recipes for the AMD optimized libraries of BLIS, libFLAME, ScaLAPACK, LibM, and FFTW will be available in the new GitHub repository (https://github.com/amd/spack). The older AMD Spack GitHub repository (https://github.com/amd/aocl-spack) is deprecated.

2. AOCL Spack recipes for BLIS, libFLAME, ScaLAPACK, LibM, and FFTW libraries are also upstreamed in the main community repository (https://github.com/spack/spack).

12.1  Setting Up AOCL Spack Environment

Complete the following steps to set up the AOCL Spack environment:

1. Clone the AMD Spack GitHub repository:

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

2. Set environment path for the Spack shell:

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

12.2  Installing AOCL Packages

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

12.2.1  Installing amdblis Spack Package

Build and install sequential BLIS:

```
$ spack install amdblis
```

Build and install BLIS with OpenMP multithreading:

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

12.2.2  Installing amdlibflame Spack Package

```
$ spack install amdlibflame
```
12.2.3 Installing amdfftw Spack Package

Install AMD FFTW Spack package:

```
$ spack install amdfftw
```

12.2.4 Installing amdscalapack Spack Package

Install AMD ScaLAPACK with AMD BLIS and libFLAME libraries:

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

12.2.5 Installing amdsparse Spack Package

Build and install AOCL-Sparse:

```
$ spack install aocl-sparse
```

12.2.6 Installing amdlibm Spack Package

Install AMD LibM Spack package:

```
$ spack install amd-libm
```

12.2.7 Installing Legacy AOCL versions

By default, Spack installs the latest versions of the AOCL libraries. However, you can install the legacy versions of the libraries by suffixing `@` followed by the desired legacy version.

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

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

12.3 Spack Commands

A few useful Spack commands to get an additional information on the Spack packages are as follows:

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display the BLIS package info and supported versions</td>
<td><code>$ spack info amdblis</code></td>
</tr>
<tr>
<td>Install BLIS</td>
<td><code>$ spack install amdblis</code></td>
</tr>
<tr>
<td>Verify installation</td>
<td><code>$ spack spec amdblis</code></td>
</tr>
<tr>
<td>Go to BLIS install directory</td>
<td><code>$ spack cd -i amdblis</code></td>
</tr>
<tr>
<td>Install other versions of amdblis package, use @&lt;version-number&gt;</td>
<td><code>$ spack install -v amdblis@2.1</code></td>
</tr>
<tr>
<td>Check supported versions, run the command</td>
<td><code>$ spack versions amdblis</code></td>
</tr>
<tr>
<td>Build and install BLIS 2.2 with OpenMP multithreading</td>
<td><code>$ spack install amdblis@2.2 threads=openmp</code></td>
</tr>
</tbody>
</table>
For more information, refer the Spack documentation (https://spack.readthedocs.io/en/latest/basic_usage.html).

BLIS installation directory contains a .spack directory comprising of the following files or directories:

### Table 14. Spack Directory Files

<table>
<thead>
<tr>
<th>File/Directory</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>.spack-build-env.txt</td>
<td>Captures the build environment details</td>
</tr>
<tr>
<td>.spack-build-out.txt</td>
<td>Captures the build output</td>
</tr>
<tr>
<td>.spec.yaml</td>
<td>Captures the installed version, arch, compiler, namespace, configure parameters, and package hash value</td>
</tr>
<tr>
<td>.repos</td>
<td>The directory containing the Spack recipe and repo namespace files</td>
</tr>
</tbody>
</table>

### 12.4 Uninstalling AOCL Packages

A sample list of commands for uninstalling AOCL Spack packages is as follows:

### Table 15. Sample Commands

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uninstall BLIS default package</td>
<td>$ spack uninstall amdblis</td>
</tr>
<tr>
<td>Uninstall libFLAME default package</td>
<td>$ spack uninstall amdlibflame</td>
</tr>
<tr>
<td>Uninstall FFTW default package</td>
<td>$ spack uninstall amdfftw</td>
</tr>
<tr>
<td>Uninstall BLIS based out of different versions</td>
<td>$ spack uninstall amdblis@2.0</td>
</tr>
<tr>
<td>Uninstall BLIS based out of hash values</td>
<td>$ spack uninstall amdblis/43reafx</td>
</tr>
<tr>
<td>Uninstall aocl-sparse default package</td>
<td>$ spack uninstall aocl-sparse</td>
</tr>
</tbody>
</table>
Chapter 13  Integration with AOCL

This section provides examples of how AOCL can be linked with High Performance Computing (HPC) and CPU intensive applications and libraries.

13.1 High-performance LINPACK Benchmark (HPL)

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

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

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

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

Setup HPL.dat before running the benchmark.

13.1.1 Configuring HPL.dat

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

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

13.1.2 Running the Benchmark

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

The HPL benchmark typically produces a better single node performance number with the following configurations depending on which generation of AMD EPYC™ processor is used:
• **2nd Gen AMD EPYC™ Processors (codenamed “Rome”)**

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

Set the following flags while building and running the tests:

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

Execute the following command to run the test:

```plaintext
mpirun -np 32 --report-bindings --map-by ppr:1:13cache,pe=4 -x OMP_NUM_THREADS=4 -x OMP_PROC_BIND=TRUE -x OMP_PLACES=cores ./xhpl
```

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

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

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

Set the following flags while building and running the tests:

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

Execute the following command to run the test:

```plaintext
mpirun -np 16 --report-bindings --map-by ppr:1:13cache,pe=8 -x OMP_NUM_THREADS=8 -x OMP_PROC_BIND=TRUE -x OMP_PLACES=cores ./xhpl
```

### 13.2 MUMPS Sparse Solver Library

MUMPS (MUltifrontal Massively Parallel Solver) is an open-source package for solving systems of linear equations of the form:

\[ Ax = b \]

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

- **Unsymmetric**
- **Symmetric positive definite**
- **General symmetric**

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

\[ A = LU \]
Where, L is a lower triangular matrix and U an upper triangular matrix.

If the matrix is symmetric then the factorization:

\[ A = LDL^T \]

Where, D is a block diagonal matrix performed.

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

1. **Analysis**
   - During an analysis, preprocessing including re-ordering and a symbolic factorization are performed. This depends on the external libs METIS, SCOTCH, and PORD (inside MUMPS source). \( A_{pre} \) denotes the preprocessed matrix.

2. **Factorization**
   - During the factorization, \( A_{pre} = LU \) or \( A_{pre} = LDL^T \), depending on the symmetry of the preprocessed matrix, is computed. The original matrix is first distributed (or redistributed) onto the processors depending on the mapping computed during the analysis. The numerical factorization is then a sequence of dense factorization on the frontal matrices.

3. **Solution**
   - The solution \( x_{pre} \) of:
     - \( LUx_{pre} = b_{pre} \) or \( LDL^T x_{pre} = b_{pre} \)
   - Where, \( x_{pre} \) and \( b_{pre} \) are the transformed solution \( x \) and right-hand side \( b \) respectively. They are associated to the preprocessed matrix \( A_{pre} \) and obtained through the forward elimination step:
     - \( Ly = b_{pre} \) or \( LDy = b_{pre} \)
   - Followed by the backward elimination step:
     - \( Ux_{pre} = y \) or \( L^T x_{pre} = y \).
   - The solution \( x_{pre} \) is finally processed to obtain the solution \( x \) of the original system \( Ax = b \).

The AOCL libraries can be integrated with the MUMPS sparse solver to perform highly optimized linear algebra operations on AMD EPYC™ processors.

### 13.2.1 Enabling AOCL with MUMPS

#### 13.2.1.1 Using Spack On Linux

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

1. Set up Spack on the target machine as explained in “Setting Up AOCL Spack Environment” on page 85.
2. Link the AOCL libraries BLIS, libFLAME, and ScaLAPACK while installing MUMPS. Use the following Spack commands to install MUMPS with:
   - gcc compiler:
     ```bash
     $ spack install mumps ^amdblis ^amdlibflame ^amdscalapack
     ```
   - aocc compiler:
     ```bash
     $ spack install mumps ^amdblis ^amdlibflame ^amdscalapack %aocc
     ```
   - To use an external reordering library (for example, METIS), run the following command:
     ```bash
     $ spack install mumps ^metis ^amdblis ^amdlibflame ^amdscalapack
     ```

13.2.1.2 On Windows

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

Prerequisites

Ensure that the following prerequisites are met:

- CMake and Ninja Makefile Generator — Ensure that Ninja is installed/updated in the Microsoft Visual Studio installation folder:
  ```
  C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\Common7\IDE\CommonExtensions\Microsoft\CMake\Ninja
  ```

  Download the latest Binary Ninja from the URL:


- Pre-built AOCL libraries for BLIS, libFLAME, and ScaLAPACK.

- If reordering library is METIS, complete the following steps:
  b. Build METIS library from the `metis` folder:
     ```
     cd SuiteSparse\metis-5.1.0
     ```
  c. Define `IDXTYPEWIDTH` and `REALTYPEWIDTH` to 32 or 64 based on the required integer size in `metis/include/metis.h`.
  d. Configure:
     ```
     cmake S . -B ninja_build_dir -G "Ninja" -DBUILD_SHARED_LIBS=OFF
     -DCMAKE_BUILD_TYPE=Release -DCMAKEVERBOSE_MAKEFILE=ON
     ```
  e. Build the project:
     ```
     cmake --build ninja_build_dir --verbose
     ```

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

Chapter 13 Integration with AOCL 91
• Boost libraries on Windows:
  – Required to read the .mtx files efficiently and quickly
  – Essential for the test application aocl_amd.cpp that links to MUMPS libraries and measures the performance for an SPD .mtx file
  – Download sources and bootstrap as instructed in the following URL:
    https://www.boost.org/doc/libs/1_55_0/more/getting_started/windows.html
  – Define BOOST_ROOT in tests/CMakeLists.txt

Building MUMPS Sources

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

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

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

The following options are enabled in the command:

- `-DENABLE_AOCL=ON`: <Enable AOCL Libraries>
- `-DENABLE_MKL=OFF`: <Enable MKL Libraries>
- `-DBUILD_TESTING=ON`: <Enable Mumps linking to test application to test>
- `-Dscotch=ON`: <Enable Metis Library for Reordering>
- `-Dopenmp=ON`: <Enable Multithreading using openmp>
- `-Dintsize64=OFF`: <Enable LP64 i.e., 32-bit integer size>
- `-DBUILD_SHARED_LIBS=OFF`: <Enable Static Library>
- `-Dparallel=ON`: <Enable Multithreading>
- `-DCMAKE_VERBOSE_MAKEFILE:BOOL=ON`: <Enable verbose build log>
- `-DCMAKE_BUILD_TYPE=Release`: <Enable Release build>
- `-DUSER_PROVIDED_BLIS_LIBRARY_PATH="<path/to/blis/lib>"`
- `-DUSER_PROVIDED_BLIS_INCLUDE_PATH="<path/to/blis/header>"`
- `-DUSER_PROVIDED_LAPACK_LIBRARY_PATH="<path/to/libflame/lib>"`
- `-DUSER_PROVIDED_LAPACK_INCLUDE_PATH="<path/to/libflame/include/header>"`
- `-DUSER_PROVIDED_SCALAPACK_LIBRARY_PATH="<path/to/scalapack/lib>"`
- `-DUSER_PROVIDED_METIS_LIBRARY="<Metis/library/with/absolute/path>"`
- `-DUSER_PROVIDED_METIS_INCLUDE_PATH="<path/to/metis/lib>"`
- `-DCMAKE_C_COMPILER="icl.exe"`
- `-DCMAKE_CXX_COMPILER="icl.exe"`
- `-DCMAKE_Fortran_COMPILER="ifort.exe"`
- `-DBOOST_ROOT="<path/to/boost_1_77_0>"`
- `-Dintsize64=OFF`

6. Toggle/Edit the options in step 5 to get:

a. Debug or Release build
b. LP64 or ILP64 libs
c. AOCL or MKL Libs
7. Build the project:

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

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

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

**Limitation**

ILP64 (or `-Dintsize64=ON`) is not supported.

**Notes:**

1. CMake build system will download the latest MUMPS tar ball ([http://mumps.enseeiht.fr](http://mumps.enseeiht.fr)) and proceed with configuration and build generation.

2. METIS Reordering has been tested. Disabling the option `-Dscotch=OFF` would enable internal reordering of MUMPS. Set the appropriate `init` parameter before calling MUMPS API in the linking test code.
Chapter 14    AOCL Tuning Guidelines

This section provides tuning recommendations for AOCL.

14.1    AOCL Dynamic

The AOCL Dynamic feature enables AOCL BLIS to dynamically change the number of threads.
This feature is enabled by default, however, it can be enabled or disabled at the configuration time using the options `--enable-aocl-dynamic` and `--disable-aocl-dynamic` respectively.

You can also specify the preferred number of threads using the environment variables `BLIS_NUM_THREADS` or `OMP_NUM_THREADS`, `BLIS_NUM_THREADS` takes precedence if both of them are specified.

The following table summarizes how the number of threads is determined based on the status of AOCL Dynamic and the user configuration using the variable `BLIS_NUM_THREADS`:

<table>
<thead>
<tr>
<th>AOCL Dynamic</th>
<th>BLIS_NUM_THREADS</th>
<th>Number of Threads Used by BLIS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disabled</td>
<td>Unset</td>
<td>Number of Cores.</td>
</tr>
<tr>
<td>Disabled</td>
<td>Set</td>
<td><code>BLIS_NUM_THREADS</code></td>
</tr>
<tr>
<td>Enabled(^a)</td>
<td>Unset</td>
<td>Number of threads determined by AOCL Dynamic.</td>
</tr>
<tr>
<td>Enabled(^a)</td>
<td>Set</td>
<td>Minimum of <code>BLIS_NUM_THREADS</code> or the number of threads determined by AOCL.</td>
</tr>
</tbody>
</table>

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

14.1.1    Limitations

The AOCL Dynamic feature has the following limitations:

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

14.2    BLIS DGEMM Multi-thread Tuning

A BLIS library can be used on multiple platforms and applications. Multi-threading adds more configuration options at runtime. This section explains the number of threads and CPU affinity settings that can be tuned to get the best performance for your requirements.
14.2.1 Library Usage Scenarios

- The application and library are single-threads:

  This is straightforward - no special instructions needed. You can export
  BLIS_NUM_THREADS=1 indicating you are running BLIS in a single-thread mode. If both
  BLIS_NUM_THREADS and OMP_NUM_THREADS are set, the former will take precedence
  over the latter.

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

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

  Example:

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

  Apart from setting the number of threads, you must pin the threads to the cores using
  GOMP_CPU_AFFINITY or numactl as follows:

  $ BLIS_NUM_THREADS=128 GOMP_CPU_AFFINITY=0-127 <./application>

  Or

  $ BLIS_NUM_THREADS=128 GOMP_CPU_AFFINITY=0-127 numactl --i=all <./application>
  $ BLIS_NUM_THREADS=128 numactl -C 0-127 --interleave=all <./test_application.x>

  Note: For the Clang compiler, it is mandatory to use OMP_PROC_BIND=true in addition to
  the thread pinning (if numactl is used). For example, for a matrix size of 200 and 32
  threads, if you run DGEMM without OMP_PROC_BIND settings, the performance
  would be less. However, if you start using OMP_PROC_BIND=true, the performance
  would improve. This problem is not noticed with libgomp using gcc compiler. For the
  gcc compiler, the processor affinity defined using numactl is sufficient.

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

  When the application is running multi-thread and number of threads are set using
  OMP_NUM_THREADS, it is mandatory to set BLIS_NUM_THREADS to one. Otherwise,
  BLIS will run in multi-thread mode with the number of threads equal to OMP_NUM_THREADS.
  This may result in a poor performance. Each application launches 64x64 threads if nested
  parallelism is enabled. So, its highly recommended to set BLIS_NUM_THREADS=1 for such
  scenarios.
• The application and library are both multi-threads:

This is a typical scenario of nested parallelism. To individually control the threading at application and at the BLIS library level, use both OMP_NUM_THREADS and BLIS_NUM_THREADS.

− The number of threads launched by the application is OMP_NUM_THREADS.
− Each application thread spawns BLIS_NUM_THREADS threads.
− To get a better performance, ensure that Number of Physical Cores = OMP_NUM_THREADS * BLIS_NUM_THREADS.

Thread pinning for the application and the library can be done using OMP_PROC_BIND:

$ OMP_NUM_THREADS=4 BLIS_NUM_THREADS=8 OMP_PROC_BIND=spread,close <./application>

OMP_PROC_BIND=spread,close

At an outer level, the threads are spread and at the inner level, the threads are scheduled closer to their master threads. This scenario is useful for a nested parallelism, where the application is running at say OMP_NUM_THREADS and each thread is calling BLIS-MT.

14.2.2 Architecture Specific Tuning

14.2.2.1 2nd and 3rd Gen AMD EPYC™ Processors

To achieve the best DGEMM multi-thread performance on 2nd Gen AMD EPYC™ processors (codenamed "Rome") and 3rd Gen AMD EPYC™ processors (codenamed "Milan"), execute one of the following commands:

Thread Size up to 16 (< 16)

OMP_PROC_BIND=spread OMP_NUM_THREADS=<NT> ./test_gemm_blis.x

Thread Size above 16 (>= 16)

OMP_PROC_BIND=spread OMP_NUM_THREADS=<NT> numactl --interleave=all ./test_gemm_blis.x

14.2.2.2 1st Gen AMD EPYC™ Processors

To achieve the best DGEMM multi-thread performance on the 1st Gen AMD EPYC™ processors (codenamed "Naples"), complete the following steps:

The header file bli_family_zen.h in the BLIS source directory \blis\config\zen defines certain macros that help control the block sizes used by BLIS.

The required tuning settings vary depending on the number threads that the application linked to BLIS runs.

Thread Size upto 16 (< 16)

1. Enable the macro BLIS_ENABLE_ZEN_BLOCK_SIZES in the file bli_family_zen.h.
2. Compile BLIS with multi-thread option as mentioned in “Multi-thread BLIS” on page 18.
3. Link the generated BLIS library to your application and execute it.

4. Run the application:

```bash
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 the multi-thread option as mentioned in “Multi-thread BLIS” on page 18.
3. Link the generated BLIS library to your application.
4. Set the following OpenMP and memory interleaving environment settings:

```bash
OMP_PROC_BIND=spread
BLIS_NUM_THREADS = x  // x> 16
numactl --interleave=all
```

5. Run the application.

   Example:

```bash
OMP_PROC_BIND=spread BLIS_NUM_THREADS=<NT> numactl --interleave=all ./test_gemm_blis.x
```

## 14.3 BLIS DGEMM Block-size Tuning

BLIS DGEMM performance is largely impacted by the block sizes used by BLIS. A matrix multiplication of large m, n, and k dimensions is partitioned into sub-problems of the specified block sizes.

Many HPC, scientific applications, and benchmarks run on high-end clusters of machines, each with multiple cores. They run programs with multiple instances through Message Passing Interface (MPI) based APIs or separate instances of each program. Depending on whether the application using BLIS is running in multi-instance mode or single instance, the specified block sizes will have an impact on the overall performance.

The default values for the block size in AMD BLIS GitHub repository ([https://github.com/amd/blis](https://github.com/amd/blis)) is set to extract the best performance for such HPC applications/benchmarks, which use single-threaded BLIS and run in multi-instance mode on AMD EPYC™ AMD “Zen” core processors. However, if your application runs as a single instance, the block sizes for an optimal performance would vary.

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

### 2nd Gen AMD EPYC™ Processors (codenamed "Rome")

1. Open the file `bli_family_zen2.h` in the BLIS source:

   ```bash
   $ cd "config/zen2/ bli_family_zen2.h"
   ```

2. For applications/benchmarks running in multi-instance mode and using multi-threaded BLIS, ensure that the macro `AOCL_BLIS_MULTIINSTANCE` is set to 0. As of AMD BLIS 2.x release,
this is the default setting. The HPL benchmark is found to generate better performance numbers using the following setting for multi-threaded BLIS:

```c
#define AOCL_BLIS_MULTIINSTANCE 0
```

### 1st Gen AMD EPYC™ Processors (codenamed "Naples")

1. Open the file `bli_cntx_init_zen.c` under the BLIS source:

   ```bash
   $ cd "config/zen/bli_family_zen.h"
   ```

2. Ensure the macro, `BLIS_ENABLE_ZEN_BLOCK_SIZES` is defined:

   ```c
   #define BLIS_ENABLE_ZEN_BLOCK_SIZES
   ```

### Multi-instance Mode

For applications/benchmarks running in multi-instance mode, ensure that the macro `BLIS_ENABLE_SINGLE_INSTANCE_BLOCK_SIZES` is set to 0. As of AMD BLIS 2.x release, following is the default setting:

```c
#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`:

```c
bli_blksz_init_easy( &blkszs[ BLIS_MC ], 144, 240, 144, 72 );
bli_blksz_init_easy( &blkszs[ BLIS_KC ], 256, 512, 256, 256 );
bli_blksz_init_easy( &blkszs[ BLIS_NC ], 4080, 2040, 4080, 4080 );
```

### Single-instance Mode

For the applications running as a single instance, ensure that the macro `BLIS_ENABLE_SINGLE_INSTANCE_BLOCK_SIZES` is set to 1:

```c
#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`:

```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 );
```
14.4 Performance Suggestions for Skinny Matrices

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

\[
C = \beta C + \alpha A*B
\]

<table>
<thead>
<tr>
<th>Dimension (Dim) of A – m x k</th>
<th>Dim(B) – k x n</th>
<th>Dim(c) – m x n</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assume row-major.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IF Dim(A) &gt;&gt; Dim(B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$BLIS_PACK_A=1 ./test_gemm_blis.x – will give a better performance.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IF Dim(A) &lt;&lt; Dim(B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$BLIS_PACK_B=1 ./test_gemm_blis.x – will give a better performance.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

14.5 AMD Optimized FFTW Tuning Guidelines

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

- Use the configure option --enable-amd-opt to build the targeted library. This option enables all the improvements and optimizations meant for AMD EPYC™ CPUs.

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

  - --enable-amd-mpifft
  - --enable-amd-mpi-vader-limit
  - --enable-amd-trans
  - --enable-amd-fast-planner
  - --enable-amd-top-n-planner
  - --enable-amd-app-opt

- When enabling the AMD CPU specific improvements with the configure option --enable-amd-opt, do not use the configure option --enable-generic-simd128 or --enable-generic-simd256.

- An optional configure option --enable-amd-trans is provided and it may benefit the performance of transpose operations in the case of very large FFT problem sizes. This feature is to be used only when running in single-thread and single instance mode.

- Use the configure option --enable-amd-mpifft to enable MPI FFT related optimizations. This is provided as an optional parameter and will benefit most of the MPI problem types and sizes.

- An optional configure option --enable-amd-mpi-vader-limit that controls enabling of AMD's new MPI transpose algorithms is supported. When using this configure option, you must set --mca btl_vader_eager_limit appropriately (current preference is 65536) in the MPIRUN command.

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

• For best performance, use the `-opatient` planner flag of FFTW.

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

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

Where, `-s` option is for speed/performance run and `icf` options stand for in-place, complex data-type, and forward transform.

• When configured with `--enable-openmp` and running multi-threaded test, set the OpenMP variables as:

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

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

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

Where, `numactl --interleave=0,1,2,3` sets the memory interleave policy on nodes 0, 1, 2, and 3.

• When running MPI FFTW test, set the appropriate MPI mapping, binding, and rank options.

For example, to run 64 MPI rank FFTW on a 64-core AMD EPYC™ processor, use:

```
mpirun --map-by core --rank-by core --bind-to core -np 64 ./mpi-bench -opatient -s icf65536
```

• Use the configure option `--enable-amd-app-opt` to enable AMD’s application optimization layer in AMD-FFTW to help uplift performance of various HPC and scientific applications. For more information, refer “AMD-FFTW” on page 104.
Chapter 15  Support

For support options, the latest documentation, and downloads refer AMD Developer Central (https://developer.amd.com/amd-aocl/).
Chapter 16  References

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

- http://www.netlib.org/scalapack/
- http://www.netlib.org/benchmark/hpl/
- https://dl.acm.org/citation.cfm?id=2764454
- https://github.com/flame/blis
- http://fftw.org/
Appendix

Check AMD Server Processor Architecture

To identify your AMD processor’s generation, perform the following steps on Linux:

1. Run the command:

```
$ lscpu
```

2. Check the values of CPU family and Model fields:
   a. For 1st Gen AMD EPYC™ Processors (codenamed “Naples”), CPU Core AMD “Zen”
      - CPU Family: 23
      - Model: Values in the range <1 – 47>
   b. For 2nd Gen AMD EPYC™ Processors (codenamed “Rome”), CPU Core AMD “Zen2”
      - CPU Family: 23
      - Model: Values in the range < 48 – 63>
   c. For 3rd Gen AMD EPYC™ Processors (codenamed “Milan”), CPU Core AMD “Zen3”
      - CPU Family: 25
      - Model: Values in the range < 1 – 15>

Application Notes

**AMD-FFTW**

- Quad precision is supported in AMD-FFTW using the AOCC v2.2 compiler (AMD clang version 10 onwards).

- Feature **AMD application optimization layer** has been introduced in AMD-FFTW to uplift the performance of various HPC and scientific applications.
  - The configure option **--enable-amd-app-opt** enables this optimization layer and must be used with the master optimization configure switch **--enable-amd-opt** mandatorily.
  - This optimization layer is supported for complex and real (r2c and c2r) DFT problem types in double and single precisions.
  - Not supported for MPI FFTs, real r2r DFT problem types, Quad or Long double precisions, and split array format.