# PLASMA Contributors' Guide

## Parallel Linear Algebra Software for Multi-core Architectures

Version 2.0

Electrical Engineering and Computer Science
**University of Tennessee**

Electrical Engineering and Computer Science
**University California, Berkeley**

Mathematical & Statistical Sciences
**University of Colorado, Denver**

Wesley Alvaro
Jakub Kurzak
Piotr Luszczek
Jack Dongarra

# Contents

# CHAPTER 1

# Introduction

This document contains all software development guidelines for the PLASMA project not documented elsewhere, in order to assure that PLASMA is a high quality software package. It is a recommended reading for new people joining the project at the participating institutions, as well as community developers.

# CHAPTER 2

# Coding Style

## 2.1 FORTRAN Style

FORTRAN means FORTRAN 77. Extensions from Fortran 90, Fortran 95, or Fortran 2003 are not allowed.

Currently PLASMA contains significant amount of FORTRAN code located in the coreblas/ directory. This code implements single-core operations on matrix tiles. In the future this code might or might not be rewritten to C for portability reasons. The advantage of keeping this code in FORTRAN is its close resemblance of LAPACK code, from which, for the most part, the code is derived. By the same token, the main coding rule, applying to the development and maintenance of this code, is that it should follow LAPACK as closely as possible. This applies to the use of whitespaces, punctuation, indentation, line breaking, the use of lower and uppercase characters, comments, variable naming, etc.

## 2.2 C Style

Only code that conforms to the ANSI C standard is allowed. The standard is commonly referred to as C89 and was ratified by ISO. One way to check for compliance is to use the following command:

```
gcc -std=c89 -W -Wall -pedantic -c plasma.c
```

Since the C89 standard does not support complex data types the following command needs to be used to remove warnings about it:

```
gcc -std=c99 -W -Wall -pedantic -c plasma.c
```

PLASMA code needs to be portable and work on Windows where the most commonly used compiler is a C++ compiler. PLASMA code must then compile with a C++ compiler. The following command will compile a C source code using the GNU C++ compiler:

```
gcc -x c++ -W -Wall -pedantic -c plasma.c
```

**No Trailing Whitespaces:** There should be no trailing whitespace characters at the end of lines, no whitespace characters in empty lines and no whitespace characters at the end of files (The last closing curly bracket should be followed by a single newline). This is easy to accomplish by using an editor that shows whitespace characters, such as Kwrite, Kate, Emacs (just use M-x delete-trailing-whitespace command). Otherwise a sed, awk, or perl "one-liner" script can be used to clean up the file before committing to the repository (e.g., tools/code_cleanup).

**Whitespace Separators:** There should be a whitespace between a C language keyword and the left round bracket and a whitespace between the right round bracket and the left curly bracket. There should be no whitespace immediately after left round bracket and immediately before right round bracket. Comas separating arguments are followed by a single space and not preceded by a space.

**End-of-line Management:** Every file should have an end-of-line character at the end unless it's a zero-length file. End-of-file character is \n (as it is on Unix including Linux; ASCII code 10). Other end-of-line schemes should not be used: Windows and DOS (\n\r – ASCII codes 10 and 13) and Mac (\r – ASCII code 13).

**Indentation:** The unit of indentation is four spaces. The left curly bracket follows the control flow statement in the same line. There is no newline between the control flow statement and the block enclosed by curly braces. The closing curly bracket is in a new line right after the end of the enclosed block.

There is no specific limit on the length of lines. Up to a 100 columns is fine. Clarity is paramount. For multi-line function calls it is recommended that new lines start in the column immediately following the left bracket.

**Tabs:** Tab characters should not be used. Tabs should always be emulated by four spaces, a feature available in almost any text editor. If that proves difficult, again, a sed, awk, or perl "one-liner" can be used to do the replacement before the commit.

**Variable Declarations:** For the most part all variables should be declared at the beginning of each function, unless doing otherwise significantly improves code clarity in a specific case.

**Constants:** Constants should have appropriate types. If a constant serves as a floating point constant, it should be written with the decimal point. If a constant is a bit mask, it is recommended that it is given in hexadecimal notation.

**printf Strings:** ANSI C concatenates strings separated by whitespace. There is no need for multiple printf calls to print a multi-line message. One printf can be used with multiple strings.

**F77 Trailing Underscore:** When calling a FORTRAN function the trailing underscore should never be used. If the underscore is needed it should be added by an appropriate conditional preprocessor definition in an appropriate header file (e.g.: core_blas.h, lapack.h).

**Special Characters:** No special characters should be used used in the code. The ASCII codes allowed in the file are between 32 and 127 and code 10 for new line.

## 2.3 Coding Practices

**Preprocessor Macros:** Conditional compilation, through the *#define* directive, should only be used for portability reasons and never for making choices that can be decided at runtime. Excessive use of the *#define* macros leads to frequent recompilations and obscure code.

**Dead Code:** There should be no dead code: no code that is never executed, no including of header files that are not necessary, no unused variables. Dead code can be justified if it serves as a comment, e.g., canonical form of optimized code. In such case the code should be in comments.

**OS Interactions:** Error checks have to follow each interaction with the OS. The code should never be terminated by the OS. In particular each memory allocation should be checked. The code cannot produce a segmentation fault.

**User Interactions:** User input needs to be checked for correctness. The user should not be able to cause undefined behavior. In particular the user should not be able to cause termination of the code by the OS.

## 2.4 Naming Convention

Any externally visible C symbols should be prefixed with PLASMA_. Following the prefix, the name should be in lower case (this will create a mixed-case name and thus will guarantee the lack of name clashes with FORTRAN interfaces that are always either all lower-case or all upper-case). For example: PLASMA_dgetrf. This is in line with C interfaces for MPI (MPI_Send), PETSc, and BLAS (BLAS_dgemm).

## 2.5 Boiler Plate text/code for Each File

Copyright, License, year, ...

## 2.6 Exceptions

As often is the case all rules have exceptions. Exceptions should only be used after consulting with the PLASMA team members.

# CHAPTER 3

# Code Generation

## 3.1 Introduction

PLASMA uses code generation to streamline the writing of similar code for multiple data types. This has been done in the past: NAG Fortran tools were used for LAPACK development and Clint Whaley's Extract for ATLAS and BLACS. Other solutions include use of C preprocessor in Goto BLAS and m4 macros in the p4 messaging system that eventually became the basis of the MPICH 1.

After looking at these tools, the PLASMA team decided to use a simpler solution: a custom Python script that resides in `tools/codegen.py`

## 3.2 Basic Usage

The usual workflow for PLASMA team when developing a new computational routine is as follows:

1. Write and debug the routine using double precision real data type using arbitrary tools (editors, compilers, etc.) without worrying about PLASMA's development tools.
2. Convert the double precision real routine so it works with double precision complex

data type.

3. Use the PLASMA's code generation script to generate single and double precision real versions and single precision complex version.

4. Compare the result of conversion with the initial version done in step 1.

Of course, this is a typical workflow so there may be others that are equally good. However, as a rule PLASMA team only maintains double precision complex version of all the computational routines: the remaining three are automatically generated with the code generation script.

The code generation step is done through the various Makefiles using the `generate` rule. A typical invocation of the script is:

```
python codegen.py core_zblas.c
```

As a result of the above command three files will be generated: `core_sblas.c`, `core_dblas.c`, and `core_cblas.c`.

## 3.3 Advanced Usage

The base code (in double precision complex) may contain annotations that help in generating code for the remaining three data types.

### 3.3.1 Complex Value Passing with `CBLAS_SADDR`

The `CBLAS_SADDR()` macro helps in dealing with old C code such as CBLAS that passes real scalars by value and complex scalars by address. Consider the matrix-matrix multiply routines:

```
double real_alpha = 1.0;
double _Complex complex_alpha = 1.0:

/* pass by value */
cblas_dgemm(col_major, trans, trans, M, N, K, real_alpha, ... );

/* pass by address */
cblas_zgemm(col_major, trans, trans, M, N, K, &complex_alpha, ... );
```

The double precision complex code in PLASMA looks like this:

```
PLASMA_Complex64_t alpha = 1.0:

cblas_zgemm( ..., CBLAS_SADDR(alpha), ... );
```

The code generation script will:

1. change `PLASMA_Complex64_t` to `PLASMA_Complex32_t` for single precision complex version of the code.
2. change `PLASMA_Complex64_t` to `double` for double precision real version of the code and will remove `CBLAS_SADDR`.
3. change `PLASMA_Complex64_t` to `float` for single precision real version of the code and will remove `CBLAS_SADDR`.

`CBLAS_SADDR` is defined as a single argument macro that returns an address of its argument so it will do the right thing for both complex versions of the code.

### 3.3.2 Conditional Code Generation

It is possible to generate code conditionally. For example Hermitian routines only make sense for complex data type:

```
#ifdef COMPLEX
void CORE_zherk(int uplo, int trans,
                int N, int K,
                double alpha, PLASMA_Complex64_t *A, int LDA,
                double beta, PLASMA_Complex64_t *C, int LDC)
{
/* ... */
}
#endif
```

On the other hand, routines specific to floating-point arithmetic make sense only for real data types. When code generation occurs, `#ifdef COMPLEX` becomes `#ifdef REAL`. More importantly, this feature requires the following lines in the original double precision complex version:

```
#undef REAL
#define COMPLEX
```

After generation, these two lines become:

```
#undef COMPLEX
#define REAL
```

So that this code works to conditionally inserting logic for real data types:

```
#ifdef REAL
double
BLAS_dfpinfo(enum blas_cmach_type cmach)
{
/* ... */
}
#endif
```

### 3.3.3 Code Dependent on Data Type

Sometimes, the code for different data types needs to be different. This can be coded in plain C without any preprossor intervention. Here is a sample:

```
if (sizeof(PLASMA_Complex64_t) == sizeof(double))
  tmult = 1; /* testing with real data types */
else
  tmult = 2; /* testing with complex data types */
```

For complex versions the `else` branch of the `if` statement is used but the compiler and `tmult` is set to 2. For single precision real version the code generating script will produce:

```
if (sizeof(float) == sizeof(float))
  tmult = 1; /* testing with real data types */
else
  tmult = 2; /* testing with complex data types */
```

and `tmult` will be set to `1`. For double precision real version the code generating script will produce:

```
if (sizeof(double) == sizeof(double))
  tmult = 1; /* testing with real data types */
else
  tmult = 2; /* testing with complex data types */
```

and `tmult` will be set to `1`. In the following example, a different code path will be taken for each of the four data types:

```
/*
 The order of the if statements is significant!
 */
if (sizeof(PLASMA_Complex64_t) == sizeof(float)) {
  printf( "single precision real\n" );
} else if (sizeof(PLASMA_Complex64_t) == sizeof(double)) {
  printf( "double precision real\n" );
} else if (sizeof(PLASMA_Complex64_t) == sizeof(PLASMA_Complex32_t)) {
  printf( "single precision complex\n" );
} else if (sizeof(PLASMA_Complex64_t) == sizeof(PLASMA_Complex64_t)) {
  printf( "double precision complex\n" );
}
```

Introduction of `if` statements might have adverse effects on performance. But modern compilers will likely remove the above `if` statements because their conditional expression is known compiled time. If preferred, the same can be accomplished *with* the preprocessor using a technique similar to the previously mentioned method. In example:

```
#define DCOMPLEX
#ifdef DCOMPLEX
...
#endif
```

or, similarly:

```
#define DCOMPLEX 1
if(DCOMPLEX){
...
}
```

Both of the above examples require only a single simple rule be added to the code generator module 3.4:

```
('SINGLE','DOUBLE','COMPLEX','DCOMPLEX')
```

## 3.4 Code Generator Modules

Code generator modules contain the rules for applying transforms and deletions for generated code. Each module can be configured to only apply to a certain pattern of files or to all files. Modules are automatically loaded from the `generator_modules` folder found in the `tools` directory.

### 3.4.1 Forward Example

Modules have a very specific structure (explained in section 3.4.2):

```
class Generator:
    name  = '''Code Generator for .c files in PLASMA''';
    match = '([a-z]+_)?($@$)([a-z0-9]*)(\_c)?(\_tile)?(\.)(c)';
    types = [
       #[[ FROM[] TYPES  ] , [FROM[0]] , [FROM[1]]          , [FROM[2]       ],
        [['ds', 'ps', 's'] , ['zc']    , ['pc','pd','pz'] , ['d','c','z'] ],
        [['ds', 'pd', 'd'] , ['zc']    , ['pc','pz','ps'] , ['s','c','z'] ],
        [['zc', 'pc', 'c'] , ['ds']    , ['ps']           , ['s','d','z'] ],
        [['zc', 'pz', 'z'] , ['ds']    , ['pc','pd','ps'] , ['s','d','c'] ],
    ];
    replacements = [
        [
           #(TYPES[0],  TYPES[1],  TYPES[2],  TYPES[3] ),
            ('SINGLE', 'DOUBLE' , 'COMPLEX', 'DCOMPLEX'),
            ('sormlq', 'dormlq' , 'cunmlq' , 'zunmlq'  ),
        ]
    ];
    deletions = [
        [
            ('CBLAS_SADDR','CBLAS_SADDR',None,None),
        ]
    ];
    replacements_all = [
        [
            ('$1$$y$$3$', '$1$@$3$'),
            (None, None) # This tuple is superfluous
        ]
    ];
    replacements_all_cleanup = [
        [
            ('$1$@$3$', '$1$$y$$3$'),
        ]
    ];
    replacements_filename = [
        [
            ('$x$$3$', '$x$$3$', '$x$$3$', '$x$$3$')
        ]
    ];
```

### 3.4.2 Description of Members

**name** Custom name for the generator

**match** Pattern for the generator to match files the @ is a special token that is replaced by the "types" of files for generation. This is a regular expression [1].

**types** This is an array of types that the generator will check for. The strings in types[i][0] will be inserted for the @ above. More specific types should come first as only the first matching type's replacements are performed.

This is the most complex part of the system. Hopefully, this real example makes it easy to understand.

Have a look at the array and then realize that:

1. 'zc', 'pz', 'z' - are all double complex types in types[3].
2. 'zc' should generate 'ds'
3. 'pz' should generate 'pc', 'pd', 'ps'
4. 'z' should generate 'c', 'd', 's'

Therefore, if a 'z' file is currently being used, for each 'c','d','s', the generator will locate those types in `types[i][0]` which are:

`types[2]`, `types[1]`, and `types[0]`, respectively.

Hence, if a 'z' file is currently being used, the rules will be applied as

$$replacements[j][3] \rightarrow replacements[j][2] \tag{3.1}$$

$$replacements[j][3] \rightarrow replacements[j][1] \tag{3.2}$$

$$replacements[j][3] \rightarrow replacements[j][0] \tag{3.3}$$

for each type, respectively.

**replacements** These are the rules for replacements. These are also regular expressions. The search needle is the column that matches the index of the source type. The replacement is the column that matches the index of the destination type. All replacements happen from top to bottom, and the list wrapping a(the) tuple(s) are only for grouping purposes.

**deletions** These are deletions. These are also regular expressions. The column that matches the destination type index is the search needle that is deleted from the file. Deletions happen from top to bottom, and the list wrapping a(the) tuple(s) are only for grouping purposes.

[1] Information on Regular Expressions

**replacements_all** These replacements occur **first** for all types. The search needle is the first column. The replacement is the secord column. These are also regular expressions. Replacements happen from top to bottom, and the list wrapping a(the) tuple(s) are only for grouping purposes.

**replacements_all_cleanup** These replacements occur **last** for all types. The search needle is the first column. The replacement is the secord column. These are also regular expressions. Replacements happen from top to bottom, and the list wrapping a(the) tuple(s) are only for grouping purposes.

**replacements_filename** These replacements apply only to a destination file's filename. These are also regular expressions. The original filename is replaced in the destination file by the resulting filename. Replacements happen from top to bottom, and the list wrapping a(the) tuple(s) are only for grouping purposes.

### 3.4.3 Special Search and Replacement Tokens

You might have noticed the special tokens in the rules as well as use of the None keyword.

**None** Search needle values that are None, or the empty string ('') are not evaluated. Replacement values can also be None to skip, but using the empty string ('') is equivalent to a deletion of patterns based on the source type (This has an obviously different effect compared to using the deletions array).

**\$#\$ (for # :** $\# \in \mathbb{N} \geq 0$**)** This is replaced with the corresponding group from the regular expression matched against the filename. It can be the empty string.

**\$x\$** This gets replaced with the type string for whatever type it is.

For example, if converting from 'z' to 's','d','c', the following is equivalent:

```
('$x$$3$','$x$$3$','$x$$3$','$x$$3$') = ('s$3$','d$3$','c$3$','z$3$')
```

**\$y\$** This is the opposite of \$x\$. It is replaced with the source type string if in the destination column and the destination type string if it is in the source column.

For example, if converting from 'z' to 's','d','c', the following is equivalent:

```
('$y$$3$','$y$$3$','$y$$3$','$y$$3$') = ('z$3$','z$3$','z$3$','[s|d|c]$3$')
```

# CHAPTER 4

# Comments

## 4.1 API Routines

Doxygen comments are used to comment these routines to automatically generate documentation (Reference Guide). These comments must be constructed in such a way that they are consistent with the other comments in the source.

### 4.1.1 Grouping Computational Routines

A routine should belong to a certain group that will cause those routines of the same group to be collected into a single Doxygen Module. This is done with the Doxygen command `@ingroup`

#### Precision

For the most part, routines are grouped by precision. This allows code generated from another source to not require any special rules.

| Routine Precision | Doxygen Group Command |
|---|---|
| `PLASMA_Complex64_t` | `@ingroup PLASMA_Complex64_t` |
| `PLASMA_Complex32_t` | `@ingroup PLASMA_Complex32_t` |
| `double` | `@ingroup double` |
| `float` | `@ingroup float` |

**Expert Interface - Asynchronous / Synchronous**

Special groups must be used for the expert API (the individual tile routines) interface consisting of Asynchronous and Synchronous functions. These groups should also abide by the precision grouping from the previous section.

| Interface | Doxygen Group Command |
|---|---|
| `..._Tile` | `@ingroup PLASMA_Complex64_t_Tile` |
| `..._Tile_Async` | `@ingroup PLASMA_Complex64_t_Tile_Async` |

### 4.1.2 Grouping Other Routines

These routines include all of the other routines, specifically those internal to the working of PLASMA.

**User Routines (Auxiliary)**

Any routine that the user should have access to falls into this category. These routines are usually prefixed with a `PLASMA_`. These routines' documentation **i**s generated for the reference manual. All of these routines are placed in the group `Auxiliary`. See section 4.1.9 for an example.

**Developer Routines (Control)**

Any routine that the user should **not** have access to falls into this category. These routines' documentation **is not** generated for the reference manual. All of these routines are currently placed in the unused group `Control`.
**Note:** While these routines are not documented for the end user, they should still be well done for your fellow developers.

### 4.1.3 Routine Documentation with LaTeXMath

The next section of comments for the routine may include the normal comments in addition to being able to take advantage of Doxygen's ability to parse LaTeXmath. You can insert LaTeXby using the Doxygen commands \$, \[, and \].

| LaTeXMath Syntax | Doxygen LaTeXMath Command |
|---|---|
| `$A\times x = b$` | `\$A\times x = b\$` |
| `\[A\times x = b\]` | `\[A\times x = b\]` |

### 4.1.4 Routine Parameters

Parameters should be specified with the following simple syntax:

| Parameter Name | Properties | Doxygen Parameter Syntax |
|---|---|---|
| A | `double*`<br>input/output | `@param[in,out] A` |
| x | `int`<br>input | `@param[in] x` |

The next line should be an indented description of the parameters role. This description can span multiple lines and can contain LaTeXformulas according to 4.1.3.

### 4.1.5 Return Values

The next section of the comments is/are the return value(s) of the routine. See the structure section (4.1.8) for reference on how to construct the return value comments.
**N**ote: The return value in the documentation must not contain spaces.

### 4.1.6 See Also Section

For a given routine the "See also" section includes the following routines:

1. The same precision, different interfaces
   (...._Tile, ...._Tile_Async),
2. The same interface, different precisions
   (PLASMA_z..., PLASMA_c..., PLASMA_d..., PLASMA_s...),
3. the same precision, the same interface, related routines
   (e.g., the solve routine for a corresponding factorization routine).

The "See also" section for the PLASMA_zgetrf() routine can serve as an example:

```
 *******************************************************************************
 *
 * @sa PLASMA_zgetrf_Tile
 * @sa PLASMA_zgetrf_Tile_Async
 * @sa PLASMA_cgetrf
 * @sa PLASMA_dgetrf
 * @sa PLASMA_sgetrf
 * @sa PLASMA_zgetrs
 *
 ******************************************************************************/
```

### 4.1.7 File Comments

Each file should have a block of comments at the top of it indicating its purpose, author(s), version, and date. The segment below is an example of how this should be done:

```
/**
 *
 * @file auxiliary.c
 *
 *  PLASMA auxiliary routines
 *  PLASMA is a software package provided by Univ. of Tennessee,
 *  Univ. of California Berkeley and Univ. of Colorado Denver
 *
 * @version 2.2.0
 * @author Jakub Kurzak
 * @author Piotr Luszczek
 * @author Emmanuel Agullo
 * @date 2009-11-15
 *
 **/
```

### 4.1.8 Comment Section Structure Summary

Comment sections should have a very specific structure. In general, the structure is such:

```
/** *********************************** ... (80 Columns wide)
 *
 * @ingroup <GROUP-NAME>
 *
 * <ROUTINE-NAME> - <DESCRIPTION>
 *
 ************************************** ...
 *
 * @param[in]     <PARAMETER-NAME>
 *    <DESCRIPTION>
 * @param[out]    <PARAMETER-NAME>
 *    <DESCRIPTION>
 * @param[in,out] <PARAMETER-NAME>
 *    <DESCRIPTION>
 *
 ************************************** ...
 *
 * @return <DESCRIPTION>
 *    \retval <VALUE> <DESCRIPTION>
 *    \retval <VALUE> <DESCRIPTION>
 *
 ************************************** ...
 *
 * @sa <SEE-ALSO>
 *
 **************************************/
```

**N**ote: Descriptions can span multiple lines.
**N**ote: A line should begin with a <SPACE><ASTERISK>
**N**ote: Sections should be separated with <SPACE><ASTERISK×79 >
**N**ote: The comment sections should begin with:
<SPACE><ASTERISK×2 ><SPACE><ASTERISK×76 >

### 4.1.9 An Actual Example : PLASMA_Version

```
/** ***************************************************************************
 *
 * @ingroup Auxiliary
 *
 *  PLASMA_Version - Reports PLASMA version number.
 *
 ******************************************************************************
 *
 * @param[out] ver_major
 *          PLASMA major version number.
 *
 * @param[out] ver_minor
 *          PLASMA minor version number.
 *
 * @param[out] ver_micro
 *          PLASMA micro version number.
 *
 ******************************************************************************
 *
 * @return
 *          \retval PLASMA_SUCCESS successful exit
 *
 *****************************************************************************/
```

# CHAPTER 5

# Miscellaneous

## 5.1 Constants

PLASMA defines a few constant parameters, such as *PlasmaTrans, PlasmaNoTrans, PlasmaUpper, PlasmaLower*, etc., equivalent of CBLAS and LAPACK parameters. The naming and numbering of these parameters follow the one of the CBLAS from Netlib (`http://www.netlib.org/blas/blast-forum/cblas.tgz`) and the C Interface to LAPACK from Netlib (`http://www.netlib.org/lapack/lapwrapc/`).

PLASMA includes a macro, *lapack_const()*, which takes PLASMA's (integer) constants and returns LAPACK's (string) constants. From the standpoint of LAPACK, only the first letter of each string is significant. Nevertheless, the macro returns meaningful strings, such as *"No transpose", "Transpose", "Upper", "Lower"), etc.*