FlexiBLAS - A flexible BLAS library with runtime exchangeable backends

Martin Köhler Jens Saak

December 20, 2013

Abstract

The BLAS library is one of the central libraries for the implementation of numerical algorithms. It serves as the basis for many other numerical libraries like LAPACK, PLASMA or MAGMA (to mention only the most obvious). Thus a fast BLAS implementation is the key ingredient for efficient applications in this area. However, for debugging or benchmarking purposes it is often necessary to replace the underlying BLAS implementation of an application, e.g. to disable threading or to include debugging symbols. In this paper we present a novel framework that allows one to exchange the BLAS implementation at run-time via an environment variable. Our concept neither requires relinkage, nor recompilation of the application. Numerical experiments show that there is no notable overhead introduced by this new approach. For only a very little overhead the framework naturally extends to a minimal profiling setup that allows one to count numbers of calls to the BLAS routines used and measure the time spent therein.

Contents

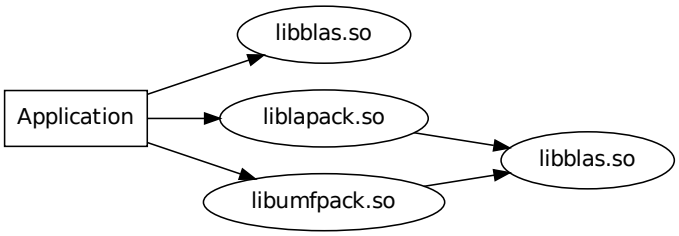

Figure 1: Shared library dependencies of an example application.

1 Introduction

The BLAS library [2, 3, 6] is one of the most important basic software libraries in scientific computing. The Netlib reference implementation defines the standard Fortran interface and its general behavior. Besides this, optimized variants for nearly every computer architecture exist. These implementations are mostly provided by the CPU or hardware vendors to get the maximum performance out of their specific devices. In addition to the vendor implementations powerful Open Source implementations like OpenBLAS [7] and the Automatically Tuned Linear Algebra Software (ATLAS) [8] exist. Besides the standard Fortran interface Netlib provides a C interface too. The so called CBLAS is normally only a wrapper to the Fortran interface which converts some data types and make the calling sequences more convenient for C programmers in the end.

Although one is only interested in using the most efficient BLAS implementation on each hardware, it is often helpful to exchange the BLAS implementation during development. This is, for example, necessary in the debugging process to turn off multithreading or to include additional debug symbols. Benchmarks, comparing different BLAS versions to find the best one for the current hardware, are other reasons to switch the BLAS implementation. Our FlexiBLAS concept accelerates this process and resolves the issues often encountered. The three main categories of these issues, the authors are aware of, are explained below.

Normally the exchange of a library is performed by relinking the program or even recompiling the entire software project. This can be a difficult and time consuming process. Unfortunately this procedure does not always solve all the problems regarding the change of the BLAS library in the whole application. The main problems are the following:

Problem 1: Dynamically Linked Libraries and their dependencies. To explain this issue, we consider an example application which uses BLAS directly for some computations and relies on LAPACK and UMFPACK [1] for more complex operations. If LAPACK and UMFPACK are linked dynamically they are usually linked against BLAS,

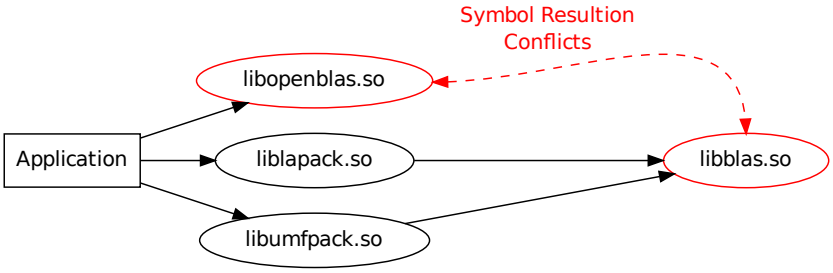

Figure 2: Wrong symbol resolution after relinking the example application.

as well, since this is necessary to resolve their internal symbols. We assume that the example application is compiled and linked dynamically using

```
gcc Applicaton.c -o Applicaton -lblas -llapack -lumfpack
```

with LAPACK and UMFPACK as shared libraries too. The dependencies between the application and the libraries are shown in Figure 1.

We observe that the BLAS library is referenced several times. First it is directly connected to the application and second it is an indirect dependency through LAPACK and UMFPACK. If one now wants to use an alternative BLAS implementation, e.g. to check if the application performs well with threading, one might link the above program against OpenBLAS via:

```
gcc Applicaton.c -o Applicaton -lopenblas -llapack -lumfpack
```

If we now take a look at the dependencies of the example application we see in Figure 2 that we integrated two different BLAS libraries. The program now tries to resolve all symbols by looking in both libraries and we can not influence which one is used. Program crashes, segmentations faults or other unwanted effects are possible and indeed often observed in this situation.

Problem 2: Incompatibilities Even though the Netlib BLAS implementation defines the standard interface and the behavior, not all available BLAS variants are 100% compatible with this interface. For example the auxiliary functions `SCABS1`/`DZABS1` are only used as tools by some BLAS routines, i.e., they are not part of the official BLAS subroutines [6], still they are provided by the Netlib reference implementation and might be called by users unaware of this fact. In order to end up with an 100% Netlib compatible interface we have to provide these functions even if they are not available in some BLAS implementations like ATLAS or Apple Accelerate. Other examples for incompatibilities are differing interfaces for functions returning complex values inside the Intel® MKL [5]. This affects `ZDOTC`, `ZDOTU`, `CDOTC` and `CDOTU`. However, there is a GNU compatible interface of the MKL in which these functions behave like the

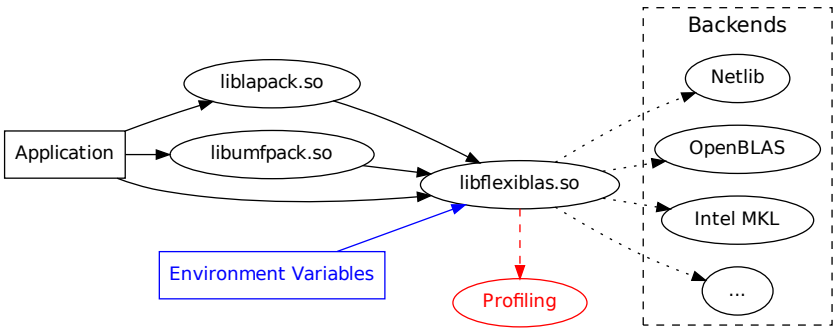

Figure 3: Application with integrated FlexiBLAS.

ones from the reference. If we use the icc/ifort compatible interface the signature of these functions changes, though. Intel® introduced an additional parameter to return the result and not as a normal return value. In turn it changes the Fortran function to a Fortran subroutine. If a program uses at least one of those functions when recompiling against different BLAS version, the source code needs to be modified to match the varying BLAS interfaces.

Problem 3: Easy Profiling In many cases it is desirable to count how often a certain BLAS function is called in a given context and how much time is spent inside this function. This type of profiling is normally enabled by special compiler options. If the BLAS library was not compiled with these options and we can not recompile it ourselves to enable them, profiling can be complicated or impossible. This is often the case when we use a vendor provided BLAS implementation like Intel® MKL or AMD ACML. Besides the recompilation, additional tools like `gprof` are necessary. Also enabling the profiling flags can considerably slow down the entire application because the compiler integrates additional code and information.

With FlexiBLAS we try to overcome all three problems by providing a wrapper library which acts as an interface between the application and the real BLAS implementation. The API of the wrapper looks exactly like the standard Netlib Fortran reference implementation and behaves like that independent of the actual implementation used. Additionally FlexiBLAS can also provide the CBLAS reference interface defined by Netlib. The underlying BLAS, the so called *backend*, is selected via configuration files or an environment variable. Figure 3 shows how the application and its dependencies connect to FlexiBLAS. Inside our wrapper a simple profiling system to analyze the function calls can be integrated very naturally and at very limited cost.

The remainder of our article is structured as follows. In the following section we describe the basic techniques behind the FlexiBLAS implementation and its usage. The computational results in Section 4 will show that the additional overhead caused by

FlexiBLAS is negligible.

2 Implementation Details

The main idea behind our wrapper library is to use the `dlopen`-mechanism specified in POSIX.1-2001 [4]. The mechanism allows one to open shared libraries or other `elf`-binaries and search for functions or symbols inside of them. This technique is often used for plugin infrastructures where parts of a program are loaded at runtime instead of linking the entire program against them. This mechanism gives us great flexibility to enhance the features of a program or exchange parts of the program without recompiling it. This corresponds to our requirements from Section 1.

The `dlopen`-mechanism consists of four functions that we will use in our implementation:

`dlopen` The `dlopen` function is an interface to the dynamic loader. It loads a shared object or an arbitrary elf-binary in the address space of the current program and returns a handle to deal with the loaded object. The shared object can be addressed either by a relative or an absolute path. If it is an absolute path it is opened directly, otherwise it searches in the default linker paths (as they are defined in `ld.so.conf` or `LD_LIBRARY_PATH`. Additionally the behavior of the symbol resolution can be influenced by a flag. Basically there are two options. The first one, `RTLD_NOW`, resolves all symbols when a shared object is opened. The second one, `RTLD_LAZY` allows one to resolve only the necessary symbols when a function out of the shared object is called. Other flags decide if the symbols of the library are integrated into the global address space or not.

`dlsym` The `dlsym` function is the look up function of the `dlopen` mechanism. It searches a given shared object handle for a desired symbol. In this way one can extract pointers to functions or variables inside the previously loaded shared library. If a symbol is not found inside the specified handle it returns a `NULL` pointer. Using two predefined library handles `RTLD_NEXT` and `RTLD_DEFAULT` one can search inside the current address space for a symbol too.

`dlclose` The `dlclose` function closes a shared library using a reference counting system. If the shared object was opened more than once the library is closed after the last reference vanishes.

`dlerror` The error handling of the `dlopen` mechanism is realized by `dlerror`. It must be called before one of the above function calls to reset the internal error flag. If an error occurs, i.e. one of the above functions returns `NULL`, the `dlerror` function returns a human readable error string.

Besides these functions the different Unix-like operating systems provide numerous extensions. We will not use any of them in order to get a portable library which should only use POSIX compatible functions.

Unfortunately the idea of using `dlopen` to extend the functionality of the program has three issues for our case which we have to solve:

```
void *__flexiblas_library;
void *call_sdot, *call_ddot,...

__attribute__((constructor)) void flexiblas_init() {
  __flexiblas_library = dlopen(getenv("FLEXIBLAS"), RTLD_NOW | RTLD_GLOBAL);
  call_sdot = dlsym(__flexiblas_library,"sdot_");
  call_ddot = dlsym(__flexiblas_library,"ddot_");
  ...
}

__attribute__((destructor)) void flexiblas_exit() {
        dlclose(__flexiblas_library);
}
```

Figure 4: Sketch of the initialization and clean up functions.

1. If a shared object is loaded using `dlopen` its symbols will not become integrated directly into the the name space of the current program. That means we have to look up each necessary symbol or function from the backend BLAS library before the program starts. In this case the user has to call an initialization function first before a BLAS function can be used. This conflicts with our goal to replace the BLAS library without changing the Netlib API because we would need to call an initialization function before we can use the first BLAS function.

2. The symbol look-up is performed at runtime rather than at compile time. This means if we only create a library containing the `dlopen`-mechanism to load the backend the linker is not able to resolve any necessary symbols at compile time. The linker is not able to detect which symbols will be integrated later on by `dlsym` from the backend. This means that the application can not be linked correctly as an executable file.

3. Some BLAS-libraries, like the Intel® MKL, consist of multiple shared object files which do not refer to all other dependent libraries. The reason behind this is, for example, to provide flexibility regarding threading models or for dealing with different pointer and integer sizes. In this case not all internal symbols of the library can be resolved when we load it by `dlopen`. The resulting error prevents us from looking up the BLAS symbols inside the library. Calling `dlopen` for all dependent libraries will not resolve the problem in every case. The reason for this is already explained in Problem 1. If the desired BLAS library is only available as a static library `dlopen` can not use it at all. Since the solution here is very much the same we count this as a similar issue.

Developing solutions for all those problems enables us to resolve all problems mentioned in Section 1 using the `dlopen`-mechanism.

```
#define BLAS_FN(rettype, name, args, callseq)\
        rettype name##_ args {\
        rettype (*fn) args ;\
        fn = call_##name;\
        if ( fn == NULL ) {\
                fprintf(stdder, #name "_not hooked, abort\n");\
                abort(); \
        }\
        return fn callseq;\
}
BLAS_FN(float, sdot, (Int *N, float *DX, Int * INCX, float *DY, Int *INCY),\
       (N,DX,INCX, DY, INCY));
BLAS_FN(double, ddot,(Int *N, double *DX, Int * INCX, double *DY, Int *INCY),\
       (N,DX,INCX, DY, INCY));
```

Figure 5: C preprocessor macro to generate the wrapper functions.

Solution to Issue 1 The first issue is solved using a property of executable files on POSIX operating systems. By defining a proper attribute for a function we can ensure that this function is executed automatically before the `main` function of a program starts. The same can be done for a function which should be executed directly after the actual program terminates. The necessary function attributes to achieve this behavior are `__attribute__((constructor))` and `__attribute__((destructor))`[1]. They are the successors of the deprecated `_init()` and `_fini()` functions. Using this technique we define an initialization and a clean up function. The initialization function reads the several environment variables and the configuration files, which define the backend BLAS library to load, as well as the behavior of FlexiBLAS. It searches for the backend library in a set of default paths and tries to open it. If this is successful it looks for every BLAS function inside the shared library and extracts the corresponding function pointers. The current address of the function is determined by `dlsym`. The addresses are stored as ordinary `void *` pointers. The correct function signature does not play a role at this time because the `void` pointer is cast to the correct one just when the function is needed. In this way we load all 147 BLAS from the backend library before the application starts. After the program terminates the cleanup function closes the backend library automatically. The pseudo code in Figure 4 gives a sketch of how this procedure is implemented.

This strategy offers two advantages: On the one hand, the library is loaded at start up and all symbols are resolved. This avoids the expensive look up every time a BLAS function is called. On the other hand, we can check if all necessary functions are provided by the given BLAS backend. If a symbol is missing we can terminate the program right before the `main` function starts. In the case of the missing `SCABS1` function in ATLAS, we include our own implementation of this function instead of terminating the program.

[1] http://gcc.gnu.org/onlinedocs/gcc-4.8.1/gcc/Function-Attributes.html

Solution to Issue 2 The second issue is solved by implementing wrapper functions for each of the 147 BLAS functions which have exactly the same function signature as the reference Netlib implementation. This allows the linker to resolve all symbols at compile time. The wrapper has to check if the corresponding BLAS symbol is loaded by `dlsym`. If not, it has to abort the program or switch to a replacement. After this check the `void *` pointer addressing the real BLAS function in the backend is cast to a function pointer with the correct function signature. Then this function is called with all given function arguments and performs the proper BLAS computations in the backend. Because all 147 wrapper functions are equally structured and have nearly the same function body expect of the their name and the function signature we package them as a C preprocessor macro. This macro is instantiated for every BLAS function. A brief implementation for such a macro is shown in Figure 5. The macro needs the return type, the name of the function, its signature and the calling sequence. In fact the calling sequence is the same as the signature but without the data types. Using a similar macro we also add aliases for all BLAS functions without the trailing underscore. The figure also shows the instantiation of the macro for `sdot` and `ddot`. In the case of `sdot` the macro expands to

```
float sdot_ (Int *N, float *DX, Int * INCX, float *DY, Int *INCY) {
  float (*fn) (Int *N, float *DX, Int * INCX, float *DY, Int *INCY);
  fn = call_sdot;
  if ( fn == NULL ) {
    fprintf(stdder, "sdot_ not hooked, abort\n");
    abort();
  }
  return fn(N, DX, INCX, DY, INCY);
}
```

The manipulated calling procedure is shown in Figure 6. The application first calls the `sdot` routine from our FlexiBLAS library which internally calls the real `sdot` implementation from the backend. The return values are transferred using the reversed path.

Besides helping the linker to resolve all symbols, we use the wrapper functions to solve the incompatibility problem described in the Introduction. Because we always define the Netlib reference interface the wrapper functions are used to rearrange the calling sequence if the backend library has some abnormalities. In this way we can use the Intel®-ifort MKL interface for `ZDOTC`, `ZDOTU`, `CDOTC` and `CDOTU` like the standard Netlib ones without changing the application. In these few cases we replace the general macro from Figure 5 by a handwritten function.

Solution to Issue 3 The problem with incomplete shared library dependencies or static libraries can not be solved directly from inside our wrapper library. The reason behind this problem is that `dlopen` can only open one shared library per call and try to resolve the missing symbols inside the library from its shared library dependencies if they are present in the current linker search path. In some cases these dependencies

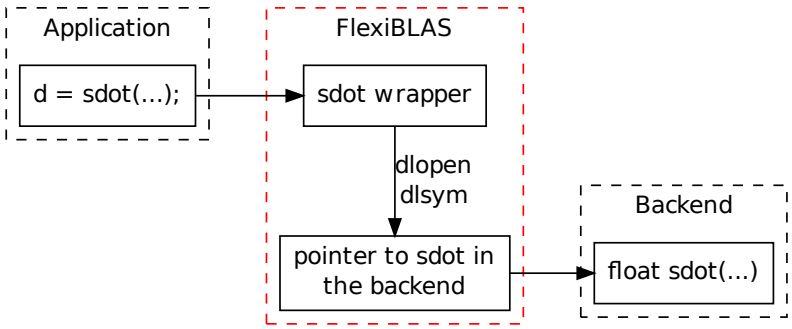

Figure 6: Calling `sdot` from an application.

```
void blas_dummy_function(){
  sdot_();
  ddot_();
  cdotc_();
  cdotu_();
  ...
}
void __flexiblas_info(struct flexiblas_info *info) {
 info->missing_scabs1 = 1;
}
```

Figure 7: Source code of the dummy library.

are not completely included in the shared object because they are resolved by a special linking sequence at compile time. The most prominent BLAS implementations with this issue are Intel® MKL and ATLAS. In the case of the MKL the implementation is split into three parts, the frontend which provides the interface to the user program, a threading middleware to change the threading model and the backend which includes all computational program code. In the case of ATLAS the Fortran standard library is not referenced if it is compiled using the GNU Fortran compiler. This is necessary to use a new version of the compiler without performing a time-consuming recompile of the library.

The issue is solved by a smart usage of the linker and a dummy library. Our first idea was to create a dummy library without own symbols only referencing the necessary BLAS library and its dependencies. Intuitively we would call:

```
gcc -shared -o libmkl_wrapper.so -lmkl_gf -lmkl_gnu_thread \
    -lmkl_core -lgomp
```

to generate a dummy library including a threaded MKL. Unfortunately, the linker does not integrate the symbols in `libmkl_wrapper.so` such that they can not be resolved

```
#define BLAS_FN_VOID_PROFILE(name, args, callseq)\
        void name args {\
        void (*fn) args ;\
        fn = call_##name;\
        double time_start = get_wtime();\
        call_counter_##name ++;\
        if ( fn == NULL ) {\
                fprintf(stdder, #name "_not hooked, abort\n");\
                abort(); \
        }\
        fn callseq;\
        time_##name = get_wtime()-time_start;\
        return;\
}
```

Figure 8: Wrapper macro with integrated profiling for a Fortran subroutine.

when we call `dlsym` on it. The solution is to create a non-static dummy function inside the library which calls or references each of the 147 BLAS symbols once. Introducing this dummy function the linker is forced to integrate all necessary symbols and their dependencies in the dummy library. If we now open the dummy library using `dlopen` it is possible to search for all BLAS symbols inside the desired backend. The knowledge about the correct calling sequence of the functions is not necessary inside the dummy function. It is enough to list all BLAS functions with following parentheses to identify the function call as shown in Figure 7.

An additional feature of introducing this dummy library is that we can integrate an *information* function. This function provides information about the used BLAS library, such as missing functions, or functions with non-standard signatures. This is used to install the correct wrappers such that our library always behaves and looks like the Netlib reference interface. The example in Figure 7 shows the information function for ATLAS. If FlexiBLAS now opens such a dummy library it calls the `__flexiblas_info` function first before it installs the wrappers. Depending on the results in the `info` structure the behavior of some wrappers is changed. At the moment this affects the aforementioned functions from the Intel® MKL and ATLAS. Beside this purpose the dummy library allows to create a dynamically loadable backend from a static object. When the linker builds the shared library out of our dummy library it integrates all necessary functions from the static library archive into our backend. It is only necessary that the static library was compiled with position independent code, i.e. with the the `-fPIC` switch enable for `gcc` or `icc`.

Profiling Employing the `dlopen` idea presented above the first two problems from the introduction have been resolved. The profiling problem has been left open up to now. Normally, if one want to profile a program one has to compile it with a set of extra compiler options. The GNU compilers, e.g. use the `-g` and `-pg` options for this purpose. This solution has two disadvantages. On the one hand, the additional

information integrated for the profiling usually makes the program run slower, and on the other hand, one requires special tools to interpret and analyze the profiling output. Especially if one uses a proprietary BLAS implementation it is not possible to recompile the library. Even in the case where access to the source code is possible, recompilation may be time consuming (e.g. automatic tuning in ATLAS takes quite a while).

For the case where one only wants to profile the usage of the BLAS library, i.e. is interested in how long a program stays inside BLAS subroutines and how often they are called, we integrate an easy profiling possibility in FlexiBLAS. Due to the fact that we already created wrapper functions for all BLAS routines we can easily and naturally extend them to count the number of calls and measure the wall-time of each function. If the BLAS symbol called is a subroutine, i.e. it returns `void`, we can modify the general macro from Figure 5 to the one shown in Figure 8. If the symbol returns a value, i.e. it is a function in the Fortran context, the macro is slightly changed. The call-counter and the accumulated time are stored in a global variable. Using the cleanup function introduced to resolve Problem 1 the results are then printed to the screen or a file once the application terminates.

CBLAS As already mentioned, our FlexiBLAS implementation is able to provide a CBLAS compatible API, too. In general the CBLAS wrapper is realized in the same way. We extend the initialization function to search for all CBLAS symbols in the backend and we provide similar wrappers for all CBLAS functions. This works in exactly the same way as for the traditional Fortran interface. But in contrast to this we can handle the situation that the backend does not include any CBLAS function at all. If this is the case we use the Netlib CBLAS wrapper with relies on the loaded Fortran BLAS backend. As an example we present the CBLAS wrapper for the `sdot` function in Figure 9. Because of the slightly more complicated internal structure of this wrappers, especially for the level 2 and 3 BLAS operations, the wrappers can not be replaced by one single generic macro like in Figure 5. Finally this behavior allows us to use both Fortran BLAS and the CBLAS in our programs independently from the capabilities of the current backend. The profiling is integrated in the same natural way as in the Fortran case. Depending on whether the backend provides a CBLAS interface or not, we measure the runtime differently. In case a CBLAS backend exists we measure both runtime and the number of function calls. If the CBLAS wrapper redirects to the Fortran backend we only count the number of function calls and the time is measured by the Fortran BLAS wrapper function. The profiling output indicates this by printing "`redirected to BLAS`" for each CBLAS call which uses the Fortran BLAS wrappers.

3 Usage

The FlexiBLAS library is used exactly like the standard Netlib BLAS implementation. That means it only needs to be linked to an application instead of the usual BLAS library. There is no need to change any source code to use FlexiBLAS. If the application

```
float cblas_sdot( const int N, const float *X,
                  const int incX, const float *Y, const int incY){
  float dot;
  #define F77_N N
  #define F77_incX incX
  #define F77_incY incY
  if ( cblas_sdot != NULL ) {
    float (*fn)(const int , const float *, const int, const float *Y,
                const int ) = cblas_sdot;
    dot = fn(N,X,incX,Y,incY);
   } else {
    dot =  sdot_( &F77_N, X, &F77_incX, Y, &F77_incY);
   }
   return dot;
}
```

Figure 9: CBLAS wrapper for `sdot`.

is dynamically linked against other libraries which also depend on BLAS, as well, one has to make sure that those libraries are linked against FlexiBLAS, as well. Otherwise we end up with the problem described in the introduction again. If all other libraries are linked statically this problem is does not exists. Instead of

```
gcc YourApplicaton.c -o YourApplicaton -lSOMEBLAS ...
```

the application is linked by calling

```
gcc YourApplicaton.c -o YourApplicaton -lflexiblas ...
```

This is the only necessary change in order to use FlexiBLAS.

Linux distributions like Debian and all its derivates (Ubuntu, Mint, ...) include a technique called "update-alternatives" which can be used to exchange the system BLAS library to FlexiBLAS without relinking any library. In this case FlexiBLAS will be the system wide default. The FlexiBLAS Debian packages make use of this feature. Other distributions like Gentoo, OpenSuse or Fedora have a similar technique. Obviously those mechanisms allow an easy exchange of the active BLAS implementation as well. In contrast to the FlexiBLAS method, however, these approaches require superuser privileges, which especially in academic software development environments, the default user usually does not have. This is possibly the most important advantage of our approach.

Selecting the BLAS backend. The behavior of the FlexiBLAS library can be influenced by a set of environment variables, a system level and a user level configuration file. All this information is read when FlexiBLAS initializes. The configuration files are optional and skipped if they do not exist. The system level configuration is stored in a file named `flexiblasrc` which normally resides in `/etc`, or in a sub-directory `etc` of

the FlexiBLAS installation prefix. The user configuration file is named `.flexiblasrc` inside the root of the user's home directory.

Each line of the configuration files has either the following syntax

```
AliasNameForTheBlasBackend | SharedObjectOfTheBackend
```

or

```
default | AliasOfTheDefaultBlasBackend
```

If the default BLAS backend should not be set, the default entry has to be removed from both configuration files. All alias names are case-insensitive. The name of the shared objects can be a simple file name, in case the backend resides in the default search path of FlexiBLAS or an absolute path if it is located somewhere else in the file system. The default search path is the directory `$(libdir)/flexiblas` inside the FlexiBLAS installation. Here `$libdir` depends on the system and hardware of the installation but should be one of `lib`, `lib32` and `lib64`. When there are multiple equally named definitions, the last one is used. An example configuration file on a system with OpenBLAS and ATLAS installed looks like:

```
Netlib   | libblas_netlib.so
OpenBLAS | libblas_openblas.so
ATLAS    | libblas_atlas.so
default  | OpenBLAS
```

The configuration files are evaluated in the following order:

1. `/etc/flexiblasrc` or `etc/flexiblasrc` from the FlexiBLAS install directory.
2. `~/.flexiblasrc` for the user defined settings.

Eventually existing aliases or default settings are overwritten in the order the configuration files are read, i.e. the settings in `~/.flexiblasrc` supersede the global one.

In order to manage the configuration files easily we provide a command line tool `flexiblas` to view and change the user's default BLAS backend. Using

```
flexiblas list
```

all available BLAS backends that are registered in any of the configuration files are listed. By calling

```
flexiblas set NameOfTheBlasBackend
```

the user can set their personal default BLAS backend in `~/.flexiblasrc`. The default entry can be removed from the user configuration file via

```
flexiblas unset
```

In addition to the configuration files the following environment variables can be used to influence the behavior of FlexiBLAS:

`FLEXIBLAS` The `FLEXIBLAS` environment variable is used to override the default backend setting from the configuration files. It can either be an alias name specified in one of the configuration files or a path to a shared library file which should be used as the backend. If it is a path-less filename, FlexiBLAS tries to find this library file in its default path. If the environment variable contains an absolute path to a file, FlexiBLAS tries to open the file directly as its backend. For example, if we want to use the OpenBLAS implementation without the dummy libraries created for FlexiBLAS we set

```
export FLEXIBLAS=/usr/lib/libopenblas.so.0
```

before our program starts. Then the initialization function tries to open the specified shared library and resolve the BLAS symbols in it. If we set the environment variable via

```
export FLEXIBLAS=libblas_atlas.so
```

it searches only inside the default search path of FlexiBLAS. This is the same behavior for shared library names as in the configuration file entries.

`FLEXIBLAS_VERBOSE` If this environment variable is set to `1` additional output information will be displayed at the start and the termination of the application. This does not affect the execution speed of the program.

`FLEXIBLAS_NOPROFILE` If this environment variable is set to `1` the profiling information will not be printed to the screen or a file. The variable is only used if FlexiBLAS is compiled with the profiling support described in Section 2.

`FLEXIBLAS_PROFILE_FILE` This environment variable defines the file the profiling output should be written to. If the variable is not set or the file cannot be opened the standard error output `stderr` is used to display the profiling result.

Add more BLAS backends. As we have seen in Section 2 some BLAS libraries can be loaded directly using the `dlopen`-mechanism. If this is not the case we have to use the dummy library workaround. The template for the dummy library can be found in `src/dummy_lib.c` of the FlexiBLAS source directory. This file contains two C functions, the first one is the dummy function to resolve all BLAS symbols. The second one called `__flexiblas_info` is the information function to tell FlexiBLAS some details about the backend and its problems. At the moment this information describes whether the complex dot products use the Intel® or the GNU calling sequence and if `SCABS1` exist. These two cases can be selected by setting the `ZDOTC_MKL` or the `SCABS1_MISSING` preprocessor macro. Basically a new backend is created by

```
gcc -shared -o yourbackend.so -O2 dummy_lib.c -lA_BLAS_IMPLEMENTATION
```

Other options like the two previously mentioned macros or runtime time paths can be added to this command line as well. This allows one to create backends from static BLAS libraries, too. Finally one can add the newly created `.so` files with an alias to one of the configuration files to use it easily.

	Netlib		Intel MKL		OpenBLAS		ATLAS	
n	no FB	with FB	no FB	with FB	no FB	with FB	no FB	with FB
$1 \cdot 10^6$	0.0021	0.0021	5.4 e-5	5.3 e-5	7.1 e-5	6.7 e-5	0.0007	0.0007
$2 \cdot 10^6$	0.0044	0.0045	0.0007	0.0007	0.0011	0.0006	0.0024	0.0024

Table 1: Runtime in seconds of `daxpy` for random input data of dimension n on x86/x86-64. (FB = FlexiBLAS)

Remark for the Profiling library. If FlexiBLAS is compiled with profiling support all file names, like the shared library, the default backend path and the configuration files, get an additional `-profile` appended. The linking step changes to

```
gcc YourApplicaton.c -o YourApplicaton -lflexiblas-profile ...
```

The name of the configuration tool changes to `flexiblas-profile`.

4 Numerical Test

The numerical tests are only meant to check if the introduced wrapper functions produce a measurable overhead, or slow down the application. Numerical differences between the results of the different BLAS implementations are not covered here. We have to rely on the fact that the effects of threading on the accuracy have been considered adequately by the maintainers of the corresponding implementations.

The runtime tests are done on two different architectures. The first one is the classic x86/x86-64 architecture (Dual-Socket Intel® Xeon® E5-2690, 2×8 Cores) running Ubuntu Linux 12.04. The second one is an IBM POWER7 based big-endian system (POWER7 3.72 GHz, 6 Cores, 4-way SMT) with Fedora Linux 19. We use typical BLAS implementations like the Netlib reference, the Intel® MKL 11, OpenBLAS 0.2.8 and ATLAS 3.10.1. On the IBM architecture we restrict to ATLAS.

In order to get a proper overview of whether or not the wrappers affect the runtime we checked them for the `daxpy`, the `dgemv` and the `dgemm` calls. This covers one operation from each BLAS level. Moreover, the benchmarks are linked against the BLAS library directly and compared to linking against our FlexiBLAS with the corresponding backend library selected using the FlexiBLAS environment variable. All tests were performed multiple times and average times are taken for each BLAS function. The `daxpy` test is only performed for two large cases because otherwise the influence of other operating systems factors, like background tasks, disturb the measurement too much. Tables 1 to 6 show the average runtime of all three benchmark examples using the different BLAS libraries. We observe that all runtimes are comparable and no mentionable differences occurred. In some cases FlexiBLAS seems to be slightly faster. A possible explanation for this behavior can be that we look up all symbols and load them at initialization instead of doing this when it becomes necessary. All other very small runtime differences are regarded as measurement errors. Furthermore we see that our technique works on a non-x86 architecture as well, with the same comparable runtime results between the classical linkage and the FlexiBLAS variant.

	Netlib		Intel MKL		OpenBLAS		ATLAS	
n	no FB	with FB	no FB	with FB	no FB	with FB	no FB	with FB
500	0.0007	0.0010	6.6 e-5	8.2 e-5	7.4 e-5	6.3 e-5	0.0002	0.0002
1 000	0.0030	0.0030	0.0002	0.0001	0.0002	0.0002	0.0007	0.0007
5 000	0.0777	0.0777	0.0110	0.0112	0.0111	0.0115	0.0123	0.0102
10 000	0.3092	0.3100	0.0557	0.0557	0.0520	0.0495	0.0380	0.0482
20 000	1.2461	1.2454	0.1826	0.1905	0.1920	0.1567	0.1910	0.1910

Table 2: Runtime in seconds of `dgemv` for random input data of dimension n on x86/x86-64.

	Netlib		Intel MKL		OpenBLAS		ATLAS	
n	no FB	with FB	no FB	with FB	no FB	with FB	no FB	with FB
500	0.4672	0.4665	0.0022	0.0022	0.0030	0.0030	0.0028	0.0028
1 000	3.7289	3.7548	0.0118	0.0149	0.0169	0.0167	0.0120	0.0120
5 000	-	-	0.8470	0.8307	0.7595	0.7469	0.9251	0.9167
10 000	-	-	6.5394	6.4372	6.0482	6.0131	7.0418	6.9967

Table 3: Runtime in seconds of `dgemm` for random input data of dimension n on x86/x86-64.

Besides the two testing systems we have checked our concept using the following platforms and BLAS libraries:

- FreeBSD 9.1 (x86-64) with ATLAS and OpenBLAS,
- NetBSD 6.1 (x86) with Netlib,
- Ubuntu 12.04 (x86-64) with ATLAS, OpenBLAS, Intel® MKL and AMD ACML
- Ubuntu 13.04 (x86) with ATLAS, MKL and OpenBLAS,
- MacOS X 10.6.8 (x86) with Apple Accelerate, ATLAS and OpenBLAS.

5 Conclusions

We have shown that it is possible to develop a wrapper library around existing BLAS libraries to exchange them easily without relinking any program. The library only uses technique described in the POSIX.1-2001 standard [4] and works on all common Unix-like operating systems and is not restricted to the x86/x86-64 platform. The experiments have shown that the additional overhead caused by our wrapper strategy can be safely ignored. A possible extension to the concept presented here will be the possibility to switch the BLAS backend during runtime. This might be an advantage if one wants to switch between sequential and threaded BLAS implementations inside a single program. A possible application for this is when the number of threads that the BLAS library uses is fixed at compile time so that we need at least two different versions of the library to switch between sequential and threaded operation. The

n	ATLAS	
	no FB	with FB
$1 \cdot 10^6$	0.0015	0.0015
$2 \cdot 10^6$	0.0043	0.0043

Table 4: Runtime in seconds of `daxpy` for random input data of dimension n on POWER7.

n	ATLAS	
	no FB	with FB
500	0.0002	0.0003
1 000	0.0006	0.0006
5 000	0.0186	0.0186
10 000	0.0709	0.0709
20 000	0.2789	0.2789

Table 5: Runtime in seconds of `dgemv` for random input data of dimension n on POWER7.

n	ATLAS	
	no FB	with FB
500	0.0067	0.0067
1 000	0.0345	0.0345
5 000	2.6224	2.6113
10 000	19.4905	19.4770

Table 6: Runtime in seconds of `dgemm` for random input data of dimension n on POWER7.

ATLAS library is a good example for this behavior. Possible implementations of this feature are currently being investigated and will be part of future versions of the library. The development of a Windows version is ongoing, whilst not being POSIX.1 compliant, a construction corresponding to the `dlopen`-family is available and will be included in future releases of FlexiBLAS.

6 Acknowledgment

The authors wish to express their gratefulness to Sven Hammarling for proofreading an earlier version of the manuscript. His comments helped to improve the quality of the presentation a lot.

References

[1] T. A. Davis, *Algorithm 832: UMFPACK V4.3—An unsymmetric-pattern multifrontal method*, ACM Trans. Math. Softw., 30 (2004), pp. 196–199.

[2] J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson, *An extended set of FORTRAN Basic Linear Algebra Subprograms*, ACM Trans. Math. Softw., 14 (1988), pp. 1–17.

[3] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling, *A set of Level 3 Basic Linear Algebra Subprograms*, ACM Trans. Math. Softw., 16 (1990), pp. 1–17.

[4] IEEE, *IEEE Std 1003.1-2001 Standard for Information Technology — Portable Operating System Interface (POSIX) System Interfaces, Issue 6*, IEEE, New York, NY, USA, 2001. Revision of IEEE Std 1003.1-1996 and IEEE Std 1003.2-1992, Open Group Technical Standard Base Specifications, Issue 6.

[5] Intel®, *Math Kernel Library (MKL)*. http://developer.intel.com/software/products/mkl/.

[6] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, *Basic linear algebra subprograms for fortran usage*, ACM Trans. Math. Softw., 5 (1979), pp. 308–323.

[7] The OpenBLAS Team. http://xianyi.github.com/OpenBLAS.

[8] R. C. Whaley, A. Petitet, and J. J. Dongarra, *Automated empirical optimization of software and the ATLAS project*, Parallel Computing, 27 (2001), pp. 3–35. Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000 (`www.netlib.org/lapack/lawns/lawn147.ps`).