HIGH PERFORMANCE PROGRAMMING
UPPSALA UNIVERSITY
SPRING 2022
EXTRA LAB: HIGH PERFORMANCE LIBRARIES

This lab is about software libraries for high performance computing. Although there are many different kinds of such libraries, we will here focus on BLAS (Basic Linear Algebra Subprograms) as our main example: we will look at how much performance you can gain by using a highly optimized BLAS library implementation compared to an unoptimized BLAS library implementation, and compared to implementing everything directly in your own code.

The following topics are included in this lab:

(1) Compiling an unoptimized BLAS library

(2) Using BLAS and CBLAS routines in your code

(3) Compiling a high performance BLAS implementation (OpenBLAS)

(4) Comparing the performance of different BLAS library implementations

(5) Using LAPACK together with BLAS for more complicated linear algebra operations

Note about extra disk space: In case some of you get into trouble with your disk quota for your home directory at the university computers, there are directories with extra space that you can use, in this directory:

`/it/slask/student/hpp/`

There you can store some of your files that take up lots of space, if you like (use the subdirectory named as your username). Note however that it is only temporary storage for this course, any data you put there will be removed after the course ends. You do not have to use the `/it/slask/student/hpp/` storage space, it is only there in case you need it.

1. Main Tasks

Task 0

The code for this task is in the `Task-0` directory.

Already in Lab 1 we looked at how object files can be combined into library files, and how a program can later link to such library files. Here we will take another

Date: November 30, 2021.

look at that, to make sure we know the basics about libraries before moving on to using different BLAS libraries.

Start by doing cd into the `Task-0/friends` directory, and look at the two c-files there. Each file defines a function, and nothing else, there is no main program here.

Compile each c-file into an object file, like this:

```
gcc -c funca.c
gcc -c funcb.c
```

Then do "`ls`" to verify that o-files have been created. Next, create a library file:

```
ar crs functions.a funca.o funcb.o
```

Then do "`ls`" to verify that the `functions.a` library file has been created. As we learned in Lab 1, we can inspect the contents of both object files and library files using the "`nm`" command. Do that: run "`nm functions.a`" and verify that it really contains the two functions from the c-files.

Now, cd into the `Task-0/mainprogram` directory and look at the main program foud there. First try to build it without linking to any library, simply like this:

```
gcc main.c
```

What happens? Do you understand why?

To make it work, you can link to the `functions.a` library file that defines the necessary functions:

```
gcc main.c ../friends/functions.a
```

Do that, then run the program and check that it works as expected.

Note that it is possible to link to libraries using either a relative path to the library file (like the `../friends/functions.a` above) or using a full path, e.g. something like "`/home/elias/test7/friends/functions.a`". Try both ways, and make sure you understand what is happening.

Task 1

The code for this task is in the `Task-1` directory.

Start by reading about BLAS (Basic Linear Algebra Subprograms) here:

http://www.netlib.org/blas/

Download the "REFERENCE BLAS" code package that is available on that page, e.g. `blas-3.8.0.tgz`. The filename extension `.tgz` is just a shorter way of writing

`.tar.gz` so you can extract the package contents in the same way as you did with the `.tar.gz` file for this lab.

Note that the reference BLAS implementation we are looking at now is not expected to give particularly high performance; the point of the reference implementation is to define what the different BLAS routines are and how they are supposed to work. So we can build the reference BLAS library and use it and get correct results, but we will probably not get very good performance. Later in this lab we will use a high-performance BLAS implementation and compare its performance to the reference BLAS implementation.

Note if using your own computer: the BLAS library built in the next step is a Fortran code so you may need to install the `gfortran` package to make this work; the library is compiled using `gfortran` rather than `gcc`.

Extract the files of the "REFERENCE BLAS" code package, cd into the directory and do "make". Now the reference BLAS library should be compiled. After doing "make", look for a file called `blas_LINUX.a` – that is the library file that was created.

The program `blastest.c` uses the BLAS routine `daxpy_()` – look at the BLAS documentation to find out what that function is doing. This overview chart can be useful for quickly looking up BLAS routines:

http://www.netlib.org/blas/blasqr.pdf

There is no "daxpy" on that chart, but there is something called `xAXPY` – the "x" there can be different things, like "d" for double precision (as in our case, daxpy), or "s" for single precision. So for double precision the function name is "`daxpy`" and for single precision (`float`) the corresponding function is "`saxpy`". This naming convention is used for all BLAS routines.

Now look at the code in `blastest.c` and make sure you understand what it is doing. It simply prepares two input vectors (x and y) and a scalar (a) and then calls the `daxpy_()` BLAS routine to compute $y \leftarrow ax + y$.

Now compile and run the program. Note that since the program uses the `daxpy_()` library routine you will need to link with a library that includes an implementation of that function. Link to the `blas_LINUX.a` library file that you built earlier. Run the program and check the output. Does it seem to produce correct results?

Using BLAS routines directly in the way we just did is a little inconventient, as you can see in the `blastest.c` code: we have a strange-looking function declaration where simple integer input parameters are expected to be passed as pointers. This is because the regular BLAS functions use Fortran-style function calls. For us as C programmers this works, but it is a little awkward.

Task 2

The code for this task is in the `Task-2` directory.

To be able to use BLAS routines in a more convenient way we can use CBLAS, that is a library that works as a simple wrapper for the ordinary BLAS functions, letting us call them in a way that is more natural in a C program.

Download and compile the CBLAS library found at the http://www.netlib.org/blas/ web page. Unpack it and cd into the resulting directory. There seems to be some problem with the testing files it tries to compile by default in the current version, to get around that we can choose to build just the library, so instead of just doing "make" we just make the `alllib` target, like this:

```
make alllib
```

Then look inside the `lib` directory, there you should see the library file `cblas_LINUX.a` that has been built.

Now look at the code in `cblastest.c` – it is doing the same thing as `blastest.c` but now using CBLAS. Instead of the peculiar Fortran-style function definition we now just include the `cblas.h` header file. The function call to the BLAS routine also looks nicer now; input arguments that are integers are just passed directly, not as pointers. Now the function call looks more like we would expect for a normal C function.

Compile the `cblastest.c` program.

Note that you will need to use the `-I` compiler option to specify an additional directory where the compiler should look for `#include` files. The CBLAS header file `cblas.h` is inside the `include` directory in the CBLAS code package, so you need to specify that path.

Example: if you have the CBLAS files in this directory:

```
/home/kim/Download/CBLAS
```

then you would specify the `-I` compiler option like this:

```
gcc -I /home/kim/Download/CBLAS/include cblastest.c
```

(and then of course also link to the necessary libraries)

The `cblastest.c` program is using the CBLAS library which in turn is using the BLAS library, so you will need to link with both the BLAS and the CBLAS libraries.

When you have succeeded to compile the `cblastest.c` program, run it and check that it works as expected. It should work exactly the same way as `blastest.c`. Does it work?

Task 3

The code for this task is in the `Task-3` directory.

In the previous tasks we have seen how you can download, compile and use a BLAS library. This is always possible, but it takes some time. If there is a BLAS library

installed on the computer you are using, you can choose to use that simply by linking with `-lblas`.

Try that for the `blastest.c` program:

```
gcc blastest.c -lblas
```

Does it work? If it does not work, then that is probably becasue there is no BLAS library installed on the computer you are using. If using your own computer you can install it by doing e.g. `dnf install blas-devel` (in Fedora) or `apt install libblas-dev` (in Ubuntu).

If you are unable to get `-lblas` to work on the computer you are using, you can instead login using ssh to one of the university's Linux servers e.g. `fredholm.it.uu.se` and try it there.

Task 4

The code for this task is in the `Task-4` directory.

Now we will compile and use a high performance BLAS library implementation called "OpenBLAS". Start by reading a bit about OpenBLAS here:

http://www.openblas.net/

https://github.com/xianyi/OpenBLAS/wiki/User-Manual

Download the OpenBLAS source code package, e.g. `OpenBLAS-0.2.20.tar.gz` and unpack it. You find the download link where the web page says "You can download this project in either zip or tar formats." (If unpacking it takes too long time, use the approach of copying to `/tmp/` as described below.)

OpenBLAS can be compiled as a multi-threaded library or as a single-threaded (serial) library. We will start by testing the serial variant. To build it as a serial library, do this:

```
make USE_THREAD=0
```

This could take a few minutes, or if some slow shared file system is used it could take a much longer time. In that case, to do the compilation faster you can copy the files to `/tmp/` which corresponds to a local hard drive, and do the compilation there, e.g. like this:

```
mkdir /tmp/elias-OpenBLAS
cp OpenBLAS-0.2.20.tar.gz /tmp/elias-OpenBLAS/
cd /tmp/elias-OpenBLAS/
tar -xzf OpenBLAS-0.2.20.tar.gz
cd OpenBLAS-0.2.20/
make USE_THREAD=0
```

Then a file called `libopenblas_nehalem-r0.2.20.a` or similar should be created, where in the filename possible instead of "nehalem" you will see some other CPU architecture name depending on which computer you are using. Note that OpenBLAS

by default is trying to give you a library optimized specifically for the computer you are using; it is not trying to make it portable.

If you compiled under `/tmp/` you should copy the resulting library file (e.g. `libopenblas_nehalem-r0.2.20.a`) to your home directory when the compilation is done, because the files under `/tmp/` may be removed, that place is only intended for temporary storage.

Now that you have the serial OpenBLAS library, link to it when you build the `blastest.c`. Does it work?

So, now we see that when we have a program like `blastest.c` or `cblastest.c` that uses BLAS library routines, we can choose to link it with different BLAS library implementations. The result should be correct for any library we choose, but the performance we get may be very different, as you will see in the following tasks.

Task 5

The code for this task is in the `Task-5` directory.

Write a program that tests and measures timings for matrix-matrix multiplication using the BLAS routine `dgemm_()`. Look up the documentation of `dgemm_()` for example at www.netlib.org.

Start from the skeleton code given in `dgemm-test.c`. For simplicity consider only square matrices, let the program use a single input argument giving the matrix size N. Then all the involved matrices A, B, C are of dimension $N \times N$.

We are here dealing with a matrix representation where each matrix is stored as a single memory block. Different conventions for how matrix elements are addressed are then possible. In particular, a matrix element A_{ij} (row i, column j) can be stored either as
`A_ij = A[i*N+j]`
or as
`A_ij = A[j*N+i]`
and when you write your own code without using any library for matrix operations you can of course choose whichever convention you want. However, when using a BLAS routine like `dgemm_()` it is necessary to take into account that BLAS routines use the second convention: `A_ij = A[j*N+i]`.

Note that the `dgemm_()` BLAS function is for general matrix-matrix multiplication, it can handle general matrices so it takes several different arguments related to the size of the matrices, so in our case the `dgemm_()` input arguments n, k and l will all be equal to N.

The `dgemm_()` input arguments lda, ldb, and ldc specify the "leading dimension" for each matrix. The "leading dimension" arguments exist to make the `dgemm_()` function even more general; they can be used to handle cases when a matrix is storead as part of a larger matrix. In our case, the "leading dimension" is simply equal to N.

If you encounter linking errors related to "gfortran", you can solve that by linking with the `-lgfortran` in addition to the BLAS library you are using.

When you have the program working and giving correct results, check what performance you get for the `dgemm_()` operation when you link to different BLAS library implementations. Try the following different cases:

- The reference BLAS implementation from Task 1.
- The default BLAS implementation on your computer (`-lblas`)
- OpenBLAS, single-threaded version
- OpenBLAS, multi-threaded version

To get a multi-threaded version of the OpenBLAS library, simply do `make` instead of `make USE_THREAD=0`.

Try different matrix sizes, from a few hundred to a few thousand. Which BLAS implementation seems to give best performance? How big is the difference compared to the time it takes for your own matrix-matrix multiplication code? Does an optimized BLAS library do it 2x faster? 10x faster? 100x faster?

Try to optimize your own matrix-matrix multiplication code as much as you can. Are you able to get close to the optimized BLAS library performance?

Task 6

The code for this task is in the `Task-6` directory.

LAPACK

LAPACK consists of a set of functions in a form similar to BLAS, the difference is that LAPACK contains more advanced linear algebra operations, and LAPACK routines usually call BLAS routines internally to do parts of their work. There are LAPACK routines many standard linear algebra computations, e.g. eigenvalue computations and solution of linear equation systems.

You can read more about LAPACK here:

http://www.netlib.org/lapack/

In this task, write a program that tests the `dsyev_()` LAPACK routine for computing eigenvalues and eigenvectors of a symmetric matrix. You can find a declaration for this function in `dsyev-test.c`. Use a program like Octave or Matlab to check that you get correct results.

Since LAPACK routines usually call BLAS routines internally, the performance of LAPACK routines can depend a lot on which BLAS implementation we are using. Test your `dsyev` test code together with different BLAS libraries, and check what performance you get. Do you see a significant difference? What about the performance compared to computing eigenvalues and eigenvectors using Octave or Matlab?