ATLAS Installation Guide *

R. Clint Whaley †

November 30, 2022

Abstract

This note provides a brief overview of ATLAS, and describes how to install it. It includes extensive discussion of common configure options, and describes why they might be employed on various platforms. In addition to discussing how to configure and build the ATLAS package, this note also describes how an installer can confirm that the resulting libraries are producing correct answers and running efficiently. Extensive examples are provided, including a full-length example showing the installation of both ATLAS and LAPACK on an example architecture.

*This work was supported in part by National Science Foundation CRI grant SNS-0551504

†`rwhaley@users.sourceforge.net`, `www.cs.utsa.edu/~whaley`

Contents

1 Introduction

This note provides a quick reference to installing and using ATLAS [17, 14, 15, 16, 20, 19]. ATLAS (Automatically Tuned Linear Algebra Software), is an empirical tuning system that produces a BLAS [5, 6, 7, 11, 12] (Basic Linear Algebra Subprograms) library which has been specifically optimized for the platform you install ATLAS on. The BLAS are a set of building block routines which, when tuned well, allow more complicated Linear Algebra operations such as solving linear equations or finding eigenvalues to run extremely efficiently (this is important, since these operations are computationally intensive). For a list of the BLAS routines, see the FORTRAN77 and C API quick references guides available in the ATLAS tarfile at:

```
ATLAS/doc/cblasqref.pdf
ATLAS/doc/f77blasqref.pdf
```

ATLAS also natively provides a few routines from the LAPACK [2] (Linear Algebra PACKage). LAPACK is an extremely comprehensive FORTRAN77 package for solving the most commonly occurring problems in numerical linear algebra. LAPACK is available as an open source FORTRAN77 package from netlib [18], and its size and complexity effectively rule out the idea of ATLAS providing a full implementation. Therefore, we add support for particular LAPACK routines only when we believe that the potential performance win we can offer make the extra development and maintenance costs worthwhile. Presently, ATLAS provides roughly 40 routines, all of which derive from our improved LU and Cholesky factorizations, which use recursive blocking. The standard LAPACK routines use statically blocked routines, which typically run slower than recursively blocked for all problem sizes. ATLAS's LU and Cholesky factorizations are based on the work of [13, 9, 10, 1, 8].

In addition to providing the standard FORTRAN77 interface to LAPACK, ATLAS also provides its own C interface, modeled after the official C interface to the BLAS [4, 3], which includes support for row-major storage in addition to the standard column-major implementations. Note that there is no official C interface to LAPACK, and so there is no general C API that allows users to easily substitute one C-interface LAPACK for another, as there is when one uses the standard FORTRAN77 API. For a list of the LAPACK routines that ATLAS natively supplies, see the FORTRAN77 and C API quick references guide available in the ATLAS tarfile at:

```
ATLAS/doc/lapackqref.pdf
```

Note that although ATLAS provides only a handful of LAPACK routines, it is designed so that it can easily be combined with netlib LAPACK in order to provide the complete library. See Section 3.1 for details.

2 Overview of an ATLAS Installation

2.1 Downloading the software and checking for known errors

The main ATLAS homepage is at:

```
http://math-atlas.sourceforge.net/
```

The software link off of this page allows for downloading the tarfile. The explicit download link is:

`https://sourceforge.net/project/showfiles.php?group_id=23725`

Once you have obtained the tarfile, you untar it in the directory where you want to keep the ATLAS source directory. The tarfile will create a subdirectory called `ATLAS`, which you may want to rename to make less generic. For instance, assuming I have saved the tarfile to `/home/whaley/dload`, and want to put the source in `/home/whaley/numerics`, I could create ATLAS's source directory (`SRCdir`) with the following commands:

```
cd ~/numerics
bunzip2 -c ~/dload/atlas3.8.0.tar.bz2 | tar xfm -
mv ATLAS ATLAS3.8.0
```

Before doing anything else, scope the ATLAS errata file for known errors/problems that you should fix/be aware of before installation:

`http://math-atlas.sourceforge.net/errata.html`

This file contains not only all bugs found, but also all kinds of platform-specific installation and tuning help.

2.2 Turn off CPU throttling when installing ATLAS

Most OSes (including Linux) now turn on CPU throttling for power management even if you are using a desktop machine. CPU throttling makes pretty much all timings completely random, and so any ATLAS install will be junk. Therefore, before installing ATLAS, turn off CPU throttling. For most PCs, you can switch it off in the BIOS (eg., on my Athlon-64 machine, I can say "No" to "Cool and Quiet" under "Power Management"). Most OSes also provide a way to switch off CPU throttling, but that varies from OS to OS. Under Fedora, at any rate, the following command seemed to work:

```
/usr/bin/cpufreq-selector -g performance
```

On my Core2Duo, cpufreq-selector only changes the parameters of the first CPU, regardless of which cpu you specify. I suspect this is a bug, because on earlier systems, the remaining CPUs were controlled via a logical link to /sys/devices/system/cpu/cpu0/. In this case, the only way I found to force the second processor to also run at its peak frequency was to issue the following as root after setting CPU0 to performance:

```
cp /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor \
   /sys/devices/system/cpu/cpu1/cpufreq/scaling_governor
```

Under MacOS or Windows, you may be able to change this under the power settings.

ATLAS config tries to detect if CPU throttling is enabled, but it may not always detect it, and sometimes may detect it after you have disabled it. In the latter case, to force the `configure` to continue regardless of the results of the CPU throttling probe, pass this flag to `configure`:

```
-Si cputhrchk 0
```

2.3 Basic Steps of an ATLAS install

An ATLAS install is performed in 5 steps, only the first two of which are mandatory. This install process is very similar to other free software installs, particularly gnu, though the fact that ATLAS does an extremely complex empirical tuning step can make the build step particularly long running. There are two directories that we will refer to constantly in this note, which indicate both the ATLAS source and build directories:

`SRCdir` : This handle should be replaced by the path to your ATLAS source directory (eg, `/home/whaley/ATLAS3.8.0`).

`BLDdir` : This handle should be replaced by the path to your ATLAS build directory (eg, `/home/whaley/ATLAS3.8.0/Linux_P4E64SSE3`).

Note that these two directories cannot be the same (i.e. you cannot build the libraries directly in the source directory). The examples in this note show the `BLDdir` being a subdirectory of the `SRCdir`, but this is not required (in fact, any directory to which the installer has read/write permission other than `SRCdir` can be used).

The ATLAS install steps are:

1. **configure** (§3): Tell the ATLAS build harness where your `SRCdir` and `BLDdir` directories are, and allow ATLAS to probe the platform to create ATLAS's `Make.inc` and `BLDdir` directory tree.
2. **build** (§4): Tune ATLAS for your platform, and build the libraries.
3. **check**[1] (§5): Run sanity tests to ensure your libraries are producing correct answers.
4. **time**[1] (§6): Run basic timing on various ATLAS kernels in order to make sure the tuning done in the **build** step has resulted in efficient implementations.
5. **install**[1] (§7): Copy ATLAS's libraries from the BLDdir to some standard location.

It is extremely important that you read Section 3 in particular, as most users will want to throw at least one flag during the `configure` step. In particular, most installers will want to set whether to build 32 or 64-bit libraries (Section 3.5.1), and fine-tune the timer used, as discussed in Section 3.4. However, for the impatient, here is the way a typical install might look (see §3 for an explanation of the `configure` flags, since they will not work on all systems); note that the characters after the `#` character are comments, and not meant to be typed in:

```
bunzip2 -c atlas3.9.x.tar.bz2 | tar xfm -     # create SRCdir
mv ATLAS ATLAS3.9.x                           # get unique dir name
cd ATLAS3.9.x                                 # enter SRCdir
mkdir Linux_C2D64SSE3                         # create BLDdir
cd Linux_C2D64SSE3                            # enter BLDdir
../configure -b 64 -D c -DPentiumCPS=2400 \   # configure command
  --prefix=/home/whaley/lib/atlas \           # install dir
```

[1] Optional step

```
  --with-netlib-lapack-tarfile=/home/whaley/dload/lapack.tgz
make build                                          # tune & build lib
make check                                          # sanity check correct answer
make time                                           # check if lib is fast
make install                                        # copy libs to install dir
```

3 The ATLAS configure step

In this step, ATLAS builds all the subdirectories of the `BLDdir`, and creates the `make` include file used in all ATLAS's Makefiles (`Make.inc`). In order to do this successfully, you inform ATLAS where your `SRCdir` and `BLDdir` are located, and pass flags which tell `configure` what type of install you want to do. The basic way to do a configure step is:

```
cd BLDdir ; SRCdir/configure [flags]
```

A complete list of flags is beyond the scope of this paper, but you can get a list of them by passing `--help` to `configure`. In this note, we will discuss some of the more important flags only. ATLAS takes two types of flags: flags that are consumed by the initial `configure` script itself begin with `--`, and flags that are passed by `configure` to a later config step begin with only a single `-`.

We first discuss flags and steps for building a full netlib library using netlib's LAPACK (§3.1), building a shared library (§3.3), changing the compilers (§3.2), and a flag (§3.2.4) to indicate that you have no FORTRAN compiler (and thus don't need any FORTRAN APIs), and changing the way ATLAS does timings (§3.4). Finally, we consider a few miscellaneous flags (§3.5), including the flag telling ATLAS whether the resulting libraries should assume a 64 or 32 bit address space (§3.5.1).

3.1 Building a full LAPACK library using ATLAS and netlib's LAPACK

ATLAS natively provides only a relative handful of the routines which comprise LAPACK. However, ATLAS is designed so that its routines can easily be added to netlib's standard LAPACK in order to get a full LAPACK library. If you want your final libraries to have all the LAPACK routines, then you just need to pass the `--with-netlib-lapack-tarfile` flag to configure, along with the netlib tarfile that you have previously downloaded. For instance, assuming you have previously downloaded the lapack tarfile to `/home/whaley/dload/lapack.tgz`, you would add the following to your `configure` flags:

```
--with-netlib-lapack-tarfile=/home/whaley/dload/lapack.tgz
```

Configure then auto-builds a `make.inc` for LAPACK to use, and builds netlib LAPACK as part of the ATLAS install process. Section 3.1.2 provides details on how this works in case it breaks down for future releases of LAPACK (it has been tested to work with netlib LAPACK 3.1.1 and 3.2.1).

3.1.1 LAPACK APIs

Note that there is no standard C API to LAPACK. Therefore, when you build the netlib LAPACK, you get only the Fortran77 API on all platforms. Various vendor libraries provide various C APIs. ATLAS provides two types of LAPACK APIs for C.

ATLAS's clapack API: ATLAS's original C interface to the LU and Cholesky-related routines is built from the `ATLAS/interfaces/lapack/C/src/` directory, and is documented in `ATLAS/doc/cblasqref.pdf`. This API is like that of the `cblas`, in that all routines take a new argument that allows matrices to be either row- or column-major. This API is difficult to extend to all of LAPACK, since the F77 LAPACK provided by netlib only handles column-major. This API uses the CBLAS enum types for F77's string arguments, and the appropriate pass-by-value or pass-by-address. This API prefixes `clapack_` to the native lapack routine name.

ATLAS's C2F API: For routines where we depend on the Fortran LAPACK for implementation, we have created a new C/LAPACK API, which can be found in `ATLAS/interfaces/lapack/C2F/src`. This API is presently undocumented. It prefixes `ATL_C2F` to the native LAPACK routine name. This interface is designed to be quick to extend, and it's main goal is to provide access to the F77 LAPACK without requiring the caller to know the arcana of calling a Fortran77 routine from C (which varies by compiler and platform). Therefore, you get a nice C interface, using enums, with correct pass-by-address/value. For every routine that takes a work argument, this API provides two routines: one with the normal name which does not take a work, where the API does the `malloc` for the user, and a second routine with `_wrk` suffixed to the name that works exactly like the F77 routine (user must pass in appropriate workspace). So, for instance the lapack routine `geqrf` takes work, and so for double precision this API provides a routine `ATL_C2Fdgeqrf` which takes no workspace-related parameters, and a `ATL_C2Fdgeqrf_wrk` which takes workspace parameters.

The C2F API is built by simple wrappers around the F77 LAPACK, and presently we have wrappers only for the QR-related routines. It is extremely straightfoward to expand this API to cover all of LAPACK, but I won't do so unless users ask for given routines, or my own research requires them (no use producing code that one using), so let me know if there are routines you'd like to see ATLAS provide via this API.

3.1.2 Details on ATLAS's LAPACK autobuild

The main thing that you need to install the netlib LAPACK is a correct `make.inc` which provides the correct compilers, flags, etc. When mixing ATLAS and netlib lapack, you need to be sure to use compatible compiler flags. ATLAS's lapack autobuild creates a `make.inc` for LAPACK to use by substituting the ATLAS options in `make.inc.example` that netlib LAPACK provides. It auto-sets the macros `FORTRAN`, `OPTS`, `NOOPT`, `LOADER`, `LOADOPTS`, and `TIMER`. If in future releases of LAPACK they remove the `make.inc.example` file, rename the above macros, or provide additional architecture-specific macros, then this autobuild process may have to be modified. In this case, you may be able to get things working by creating a compatible `make.inc` yourself after the ATLAS configure step, but before the build step. You can see the Makefile targets for autobuilding netlib's ATLAS in `BLDdir/interfaces/lapack/C2F/Makefile`.

3.1.3 Obtaining netlib's LAPACK

You can download the LAPACK reference implementation from `www.netlib.org/lapack/`.
As of this writing, the newest LAPACK tarfile was `http://www.netlib.org/lapack/lapack.tgz`.

For more standard information on LAPACK, please scope the following URLs:

- `http://www.netlib.org/lapack/`
- `http://www.netlib.org/lapack/lawn81/index.html`
- `http://www.netlib.org/lapack/lawn41/index.html`
- `http://www.netlib.org/lapack/release_notes.html`
- `http://www.netlib.org/lapack/lug/index.html`

3.2 Changing the compilers and flags that ATLAS uses for the build

ATLAS defines eight different compilers and associated flag macros in its `Make.inc` which are used to compile various files during the install process. ATLAS's `configure` provides flags for changing both the compiler and flags for each of these macros. In the following list, the macro name is given first, and the configure flag abbreviation is in parentheses:

1. `XCC` (`xc`): C compiler used to compile ATLAS's build harness routines (these never appear in any user-callable library)
2. `GOODGCC` (`gc`): gcc with any required architectural flags (eg. `-m64`), which will be used to assemble cpp-enabled assembly and to compile certain multiple implementation routines that specifically request `gcc`
3. `F77` (`if`): FORTRAN compiler used to compile ATLAS's FORTRAN77 API interface routines.
4. `ICC` (`ic`): C compiler used to compile ATLAS's C API interface routines.
5. `DMC` (`dm`): C compiler used to compile ATLAS's generated double precision (real and complex) matmul kernels
6. `SMC` (`sm`): C compiler used to compile ATLAS's generated single precision (real and complex) matmul kernels
7. `DKC` (`dk`): C compiler used to compile all other double precision routines (mainly used for other kernels, thus the `K`)
8. `SKC` (`sk`): C compiler used to compile all other single precision routines (mainly used for other kernels, thus the `K`)

It is almost never a good idea to change `DMC` or `SMC`, and it is only very rarely a good idea to change `DKC` or `SKC`. For ATLAS 3.8.0, all architectural defaults are set using gcc 4.2 only (the one exception is MIPS/IRIX, where SGI's compiler is used). In most cases, switching these compilers will get you worse performance and accuracy, even when you are *absolutely* sure it is a better compiler and flag combination! In particular we tried the Intel compiler `icc` (called `icl` on Windows) on Intel x86 platforms, and overall performance was lower than `gcc`. Even worse, from the documentation `icc` does not seem to have any firm IEEE floating point compliance unless you want to run so slow that you could compute it by hand faster. This means that whenever `icc` achieves reasonable performance, I have no idea if the error will be bounded or not. I could not obtain access to `icc` on the Itaniums, where `icc` has historically been much faster than `gcc`, but I note that the performance of

`gcc4.2` is much better than `gcc3` for most routines, so `gcc` may be the best compiler there now as well.

There is almost never a need to change `XCC`, since it doesn't affect the output libraries in any way, and we have seen that changing the kernel compilers is a bad idea. However, what if you yourself use a non-gnu compiler, like Intel's `icc` or `ifort`, then what you need to do is tell ATLAS to compile its interface routines with your compilers, which is discussed in Section 3.2.1. Another common problem is that your OS has been built with an older `gcc` whose libraries are incompatible with `gcc 4.2`. In this case, creating an executable with `gcc4.2` can cause problems, and so what you want to do is keep `gcc3` as you default compiler (compiling ATLAS interface routines with it, as well as using it for all linking) but compile the ATLAS kernel routines with `gcc4`. This case is discussed in Section 3.2.2. For those who insist on monkeying with other compilers, Section 3.2.3 gives some guidance. Finally installing ATLAS without a FORTRAN compiler is discussed in Section 3.2.4.

3.2.1 Changing ATLAS interface compilers to match your usage

As mentioned, ATLAS typically gets its best performance when compiled with `gcc` using the flags that ATLAS automatically picks for your platform (this assumes you are installing on a system that ATLAS provides architectural defaults for). However, you can vary the interface (API) compilers without affecting ATLAS's performance. Since most compilers are interoperable with `gcc` this is what we recommend you do if you are using a non-default compiler. Note that almost all compilers can interoperate with `gcc`, though you may have to throw some special flags (eg., `/iface:cref` for MSVC++).

The configure flags to override the C interface compiler and flags are:

```
-C ic <C compiler> -F ic '<compiler flags>'
```

The configure flags to override the FORTRAN interface compiler and flags are:

```
-C if <FORTRAN compiler> -F if '<compiler flags>'
```

A few example will help here. If I wanted to use Intel's FORTRAN and C compilers under windows on a P4, I could issue:

```
-C if ifort -F if '-O2 -fltconsistency -nologo' \
-C ic icl -F ic '-QxN -O3 -Qprec -fp:extended -fp:except -nologo -Oy'
```

On the same system, if I wanted to use Intel for FORTRAN and MSVC++ for C:

```
-C if ifort -F if '-O2 -fltconsistency -nologo' \
-C ic icl -F ic '-Oy -Ox -arch:SSE2 -nologo'
```

For Windows, we can note a couple of things. First, while these flags are straight from the Windows compiler documentation, we have replaced the Windows '`/`' flag character with the Unix '`-`' flag character. This is because ATLAS doesn't call native Windows compilers directly, but rather calls a wrapper routine that makes these compilers work with make like a standard Unix compiler. The second thing to notice is that we don't have to say to use the `/iface:cref` flag, because this same wrapper always throws this flag (ATLAS does not work with the other rather bizarre naming strategies).

For a non-Windows example, assume you use the Sun Workshop compilers available under Solaris. You can instruct `configure` to use them for building the APIs rather than the gnu compilers with something like:

```
-C if f77 -F if '-dalign -native -xO5' \
-C ic cc -F ic '-dalign -fsingle -xO5 -native'
```

3.2.2 Compiling ATLAS with gcc 4.2 when your OS uses an incompatible gcc

As previously mentioned, `gcc4.2` is what the architectural defaults are built for, and previous versions are likely to hurt your performance. For systems with `gcc4.1` (the worst-performing `gcc` for x86 machines), you can usually just install `gcc4.2`, and change your path so that `gcc4.2` is your default compiler. However, between major releases the `gcc` system libraries change too much for this to work right. Therefore, if your OS was built with `gcc3`, for example, what will often happen is that executables built with `gcc4` will not be able to run, unless you fiddle with your `LD_LIBRARY_PATH` so that the `gcc4` libraries are found before those of `gcc3`. However, if you do this, then often `gcc3`-built objects, which include the majority of things you use every day (eg., editors), won't run because they find the `gcc4` libraries instead of the expected libs from `gcc3`!

Therefore, you don't want to make `gcc4.2` your default compiler, but you want to have ATLAS use it to compile all the kernel routines, while compiling interface routines and doing any linking with `gcc3`. To do this, leave the system `gcc` as the default one in your path, but pass the following flag to `configure`:

```
-Ss kern <path to gcc4.2>
```

This tells ATLAS to use all non-kernel compilers as normal, but to change all kernel compilers to the given compiler. Therefore, if I have installed `gcc4.2` on my `gcc3`-built OS in my own home area at `/home/whaley/local/gcc42`, I would add something like:

```
-Ss kern /home/whaley/local/gcc42/bin/gcc
```

3.2.3 Rough guide to overriding ATLAS's compiler choice/changing flags

Previous sections have discussed the more useful cases of overriding ATLAS's compiler and flags, which typically leave ATLAS's kernel compilers alone. Users often wish to add flags or change arbitrary compilers, however. This is rarely a good idea, and almost always provides reduced performance. However, you can do it. You can find more details by passing `--help` to `configure`.

As previously mentioned (§3.2.1), you can specify what compiler (flag setting) to override by passing the appropriate abbreviation to the `-C` (`-F`) configure flags in order to change the compiler (compiler flags). For example, you would pass `-C if` to override interface FORTRAN compiler. `configure` also supports appending certain compiler flags, so that user flags are simply added to the defaults that ATLAS uses. This is done:

```
-Fa <abbr> '<comp flags to append>'
```

where `<abbr>` is one of:

- One of the already discussed compiler abbreviations (eg, `xc`, `ic`, `if`, `sk`, `dc`, `sm` or `dm`)
- `al`: all compilers (including FORTRAN) except `GOODGCC`
- `alg` all compilers (including FORTRAN) including `GOODGCC`
- `ac`: all C compilers except `GOODGCC`
- `acg`: all C compilers including `GOODGCC`

Therefore, by passing the following to `configure`:

```
-Fa acg '-DUsingDynamic -fPIC'
```

We would have all C routines compiled with `-fPIC`, and also have the macro `UsingDynamic` defined (ATLAS does not use this macro, this is for example only).

The compiler overriding flag -C can also take the abbreviation `ac` which will override all C compilers except `GOODGCC` with the given C compiler. There is currently no flag to override `GOODGCC` on the command line, so if you need to do this, you will need to edit the output `Make.inc` after configure.

As an example, if I want to use SunOS's `f77` rather than `gfortran`, I could pass the following compiler and flag override:

```
-C if f77    -F if 'dalign -native -xO5'
```

IMPORTANT NOTE: If you change the default flags in any way for the kernel compilers (even just appending flags), you may reduce performance. Therefore once your build is finished, you should make sure to compare your achieved performance against what ATLAS's architectural defaults achieved. See Section 6.1 for details on how to do this. If your compiler is a different version of `gcc`, you may also want to tell ATLAS not to use the architectural defaults, as described in Section 3.5.4.

3.2.4 Installing ATLAS when you don't have access to a FORTRAN compiler

By default, ATLAS expects to find a FORTRAN compiler on your system. If you cannot install a FORTRAN compiler, you can still install ATLAS, but ATLAS will be unable to build the FORTRAN77 APIs for both BLAS and LAPACK. Further, certain tests will not be able to even compile, as their testers are at least partially implemented in FORTRAN. To tell ATLAS you wish to install w/o a FORTRAN compiler, simply add the flag:

```
--nof77
```

to your `configure` command.

`IMPORTANT NOTE`: When you install ATLAS w/o a FORTRAN compiler, your build step will end with a bunch of `make` errors about being unable to compile some FORTRAN routines. This is because the `Makefile`s always attempt to compile the FORTRAN APIs: they simply continue the install if they don't succeed in building them. So, just because you get a lot of `make` messages about FORTRAN, don't assume your library is messed up. As long as `make check` and `make time` say your `-nof77` install is OK, you should be fine.

3.3 Building dynamic/shared libraries

ATLAS natively builds static libraries (i.e. libs that usually end in '`.a`' under Unix and '`.lib`' under windows). ATLAS always builds such a library, but it can also optionally be requested to build a dynamic/shared library (typically ending in .so for Unix or .dll windows) as well. In order to do so, you must tell ATLAS up front to compile with the proper flags (the same is true when building netlib's LAPACK, see §3.1 for more details). Assuming you are using the gnu C and FORTRAN compilers, you can add the following commands to your `configure` command:

```
-Fa alg -fPIC
```

to force ATLAS to be built using position independent code (required for a dynamic lib). If you use non-gnu compilers, you'll need to use -Fa to pass the correct flag(s) to append to force position independent code for each compiler (don't forget the gcc compiler used in the index files).

After your build is complete, you can `cd` to your `OBJdir/lib` directory, and ask ATLAS to build the `.so` you want. If you want all libraries, including the FORTRAN77 routines, the target choices are:

shared : create shared versions of ATLAS's sequential libs

ptshared : create shared versions of ATLAS's threaded libs

If you want only C routines (eg., you don't have a FORTRAN compiler):

cshared : create shared versions of ATLAS's sequential libs

cptshared : create shared versions of ATLAS's threaded libs

Note that this support for building dynamic libraries is new in this release, and not well debugged or supported, and is much less likely to work for non-gnu compilers.

IMPORTANT NOTE: Since gcc uses one less integer register when compiling with this flag, this could potentially impact performance of the architectural defaults, but we have not seen it so far. Therefore, do not throw this flag unless you want dynamic libraries. If you want both static and dynamic libs, the safest thing is probably to build ATLAS twice, once static and once dynamic, rather than getting both from a dynamic install.

3.4 Changing the way ATLAS does timings

By default ATLAS does all timings with a CPU timer, so that the install can be done on a machine that is experiencing relatively heavy load. However, CPU time has very poor resolution, and so this makes the timings less repeatable and provides for only a rough idea of overall performance. Therefore, if you are installing ATLAS on a machine which is not heavily loaded, you will want to improve your install by instructing ATLAS to use one of its higher resolution wall timers.

For x86 machines, ATLAS has access to a cycle accurate wall timer, assuming you are using `gcc` as your interface compiler (we use `gcc`'s inline assembly to enable this timer – under Linux, Intel's `icc` also supports this form of inline assembly). ATLAS needs to be able to translate the cycle count returned by this function into seconds, so you must pass your machine's clock rate to ATLAS. In order to do this, you add the following flags to your configure flags:

```
-D c -DPentiumCPS=<your Mhz>
```

So, for my 2.4Ghz Core2Duo, I would pass:

```
-D c -DPentiumCPS=2400
```

If you are not on an x86 machine, or if your interface compiler is not `gcc` (or `icc` if on Linux), then you cannot use the above cycle-accurate wall timer. However, wall time is still much more accurate than CPU time, so you can indicate ATLAS should use its wall timer for the install by passing the flag:

```
-D c -DWALL
```

Note that on Windows XP/NT/2000, this should still get you a cycle-accurate walltime, since it calls some undocumented Windows APIs that purport to do so. For Solaris, the high resolution timer `gethrtime` will be used. For all other OSes, this will call a standard wall timer such as `gettimeofday`, which is still usually much more accurate than the CPU timer.

3.5 Various other flags

3.5.1 Changing pointer bitwidth (64 or 32 bits)

Most modern platforms allow for compiling libraries to handle either 32 or 64 bit address spaces. On the x86, this selection strongly affects the ISA used (eg., whether to use IA32 or x86-64). The x86-64 ISA, with 16 rather than 8 registers, is more amenable to optimization than the IA32, so if the user has no preference, 64-bit pointers are recommended. If ATLAS's guess is not correct, you can tell configure what address space to build for. In order to force 32-bit pointer width, pass the flag:

```
-b 32
```

and in order to force 64 bit pointers, pass:

```
-b 64
```

(the `b` stands for bitwidth).

This tells ATLAS to throw the appropriate compiler flags for compilers it knows about, as well as effecting various configure probes. Therefore, if you override ATLAS's compiler choices, be sure that you give the correct flags to match this setting.

3.5.2 Changing configure verbosity

`configure` does a series of architectural probes to figure out how to do an install on your system. Many of the probes that are run don't produce output during the configure step. You can tell `configure` that you want to see more output by cranking up the verbosity. Presently, maximum verbosity is enabled by adding the flag:

```
-v 2
```

3.5.3 Controlling where ATLAS will move files to during install step

ATLAS supplies some flags to control where ATLAS will move files to when you do the `make install` step (§2). These flags are taken from gnu `configure`, and they are:

- `--prefix=<dirname>` : Top level installation directory. include files will be moved to `<dirname>/include` and libraries will be moved to `<dirname>/lib`. Default: `/usr/local/atlas`
- `--incdir=<dirname>` : Installation directory for ATLAS's include files. Default: `/usr/local/atlas/include`.
- `--incdir=<dirname>` : Installation directory for ATLAS's libraries. Default: `/usr/local/atlas/lib`.

3.5.4 Telling ATLAS to ignore architectural defaults

Architectural defaults are partial results of past searches when the compiler and architecture are known. They allow you skip the full ATLAS search, which makes install time much quicker. They also ensure that you have good results, since they typically represent several searches and/or user intervention into the usual search so that maximum performance is found. This doesn't typically mean a huge performance difference, since the empirical search usually does an adequate job, but it often provides a few extra percentage points of performance. Also, occasionally the empirical search will, due to machine load or other timing problems, produce inadequate code, and using the architectural defaults prevents this from happening.

By default, ATLAS automatically uses the architectural defaults anytime it has results for the given architecture and compiler. However, the compiler detection is based on the compiler name, not version, and so ATLAS's architectural defaults for gnu `gcc4.2` might not be best for `gcc3` or apple's `gcc`, etc, even though `configure` would use the architectural defaults in such cases.

So, there are times when you want to tell ATLAS to ignore any architectural defaults it might have. Common reasons include the fact that you have overridden the compiler flags ATLAS uses, or are using an earlier version of the supported compiler. In these cases, the best idea is often to install both with and without the architectural defaults, and compare timings. If both your installs (homegrown-compiler/flags+archdef, homegrown-compiler/flags+search) are slower than the architectural defaults using the default compiler, you should probably install the default compiler. However, if your results are largely the same, you know your changes haven't depressed performance and so it is OK to use the generated libraries (see Section 6 for details on timing an ATLAS install). If your timing results are substantially better, and you haven't enabled IEEE-destroying flags, you should send your improved compiler and flags to the ATLAS team!

To force ATLAS to ignore the architectural defaults (and thus to perform a full ATLAS search), pass the following flags to `configure`:

```
-Si archdef 0
```

4 The ATLAS build step

This is the step where ATLAS performs all its empirical tuning, and then uses the discovered kernels to build all required libraries. It uses the `BLDdir` created by the configure step, and is invoked from the `BLDdir` with the `make build` command, or simply by `make`. This step can be quite long, depending on your platform and whether or not you use architectural defaults. For a system like the Core2Duo with architectural defaults, the build step may take 10 or 20 minutes, while in order to complete a full ATLAS search on a slower platform (eg. MIPS) could take anywhere between a couple of hours and a full day.

5 The ATLAS check step

In this optional step, ATLAS runs various testers in order to make sure that the generated library is not producing completely bogus results. For each precision, ATLAS runs the

standard BLAS testers (both C and F77 interface), and then various of ATLAS's homegrown testers that appear in `ATLAS/bin`. If you have installed without a FORTRAN compiler, then the standard BLAS testers cannot be run (the standard BLAS testers, downloadable from netlib, require FORTRAN even to test the C interface), and so your testing will be less comprehensive.

There are two possible targets, `check` which tests ATLAS's serial routines, and `ptcheck` which check the parallel routines. You cannot run `ptcheck` if you haven't installed the parallel libraries. This step is invoked from `BLDdir` by typing:

```
make check      # test serial routines
make ptcheck    # check parallel routines
```

Both of these commands will first do a lot of compilation, and then they will finish with results such as:

```
core2.home.net. make check
................................................
..... A WHOLE LOT OF COMPILATION AND RUNNING ......
................................................
DONE BUILDING TESTERS, RUNNING:
SCOPING FOR FAILURES IN BIN TESTS:
fgrep -e fault -e FAULT -e error -e ERROR -e fail -e FAIL \
        bin/sanity.out
8 cases: 8 passed, 0 skipped, 0 failed
4 cases: 4 passed, 0 skipped, 0 failed
8 cases: 8 passed, 0 skipped, 0 failed
4 cases: 4 passed, 0 skipped, 0 failed
8 cases: 8 passed, 0 skipped, 0 failed
4 cases: 4 passed, 0 skipped, 0 failed
8 cases: 8 passed, 0 skipped, 0 failed
4 cases: 4 passed, 0 skipped, 0 failed
DONE
SCOPING FOR FAILURES IN CBLAS TESTS:
fgrep -e fault -e FAULT -e error -e ERROR -e fail -e FAIL \
        interfaces/blas/C/testing/sanity.out | \
                fgrep -v PASSED
make[1]: [sanity_test] Error 1 (ignored)
DONE
SCOPING FOR FAILURES IN F77BLAS TESTS:
fgrep -e fault -e FAULT -e error -e ERROR -e fail -e FAIL \
        interfaces/blas/F77/testing/sanity.out | \
                fgrep -v PASSED
make[1]: [sanity_test] Error 1 (ignored)
DONE
make[1]: Leaving directory `/home/whaley/TEST/ATLAS3.7.36.0/obj64'
```

Notice that the `Error 1 (ignored)` commands come from `make`, and they indicate that `fgrep` is not finding any errors in the output files (thus this `make` output does not represent the finding of an error). When true errors occur, the lines of the form

```
8 cases: 8 passed, 0 skipped, 0 failed
```

will have non-zero numbers for `failed`, or you will see other tester output discussing errors, such as the printing of large residuals.

As mentioned, this is really sanity checking, and it runs only a few tests on a handful of problem sizes. This is usually adequate to catch most blatant problems (eg., compiler producing incorrect output). More subtle or rarely-occurring bugs may require running the LAPACK and/or full ATLAS testers. The ATLAS developer guide [21] provides instructions on how to use the full ATLAS tester, as well as help in diagnosing problems. The developer guide is provided in the ATLAS tarfile as `ATLAS/doc/atlas_devel.pdf`

6 The ATLAS time step

In this optional step, ATLAS times certain kernel routines and reports their performance as a percentage of clock rate. Its purpose is to provide a quick way to ensure that your install has resulted in a library that obtains adequate performance. If you are installing using architectural defaults, this step will print a timing comparison against the performance that the ATLAS maintainer got when creating the architectural defaults. To invoke this step, issue the following command in your `BLDdir`:

```
make time
```

In Figure 1 we see a typical printout of a successful install, in this case ran on my 2.4Ghz Core2Duo. The `Refrenc` columns provide the performance achieved by the architectural defaults when they were originally created, while the `Present` columns provide the results obtained using the new ATLAS install we have just completed. We see that the `Present` columns wins occasionally (eg. single precision real `kSelMM`), and loses sometimes (eg. single precision complex `kSelMM`), but that the timings are relatively similar across the board. This tells us that the install is OK from a performance angle.

As a general rule, performance for both data types of a particular precision should be roughly comparable, but may vary dramatically between precisions (due mainly to differing vector lengths in SIMD instructions).

The timings are normalized to the clock rate, which is why the clock rate of both the reference and present install are printed. It is expected that as clock rates rise, performance as a percent of it may fall slightly (since memory bus speeds do not usually rise in exact lockstep). Therefore, if I installed on a 3.2Ghz Core2Duo, I would not be surprised if the `Present` install lost by a few percentage points in most cases.

True problems typically display a significant loss that occurs in a pattern. The most common problem is from installing with a poor compiler, which will lower the performance of most compiled kernels, without affecting the speed of assembly kernels. Figure 2 shows such an example, where `gcc 4.1` (a terrible compiler for floating point arithmetic on x86 machines) has been used to install ATLAS on an Opteron, rather than `gcc 4.2`, which was the compiler that was used to create the architectural defaults. Here, we see that the present machine is actually slower than the machine that was used to create the defaults, so if anything, we expect it to achieve a greater percentage of clock rate. Indeed, this is more or less true of the first line, `kSelMM`. On this platform, `kSelMM` is written totally in assembly, and `BIG_MM` calls these kernels, and so the `Present` results are good for these rows. All the other rows show kernels that are written in C, and so we see that the use of a bad compiler

```
NAMING ABBREVIATIONS:
   kSelMM : selected matmul kernel (may be hand-tuned)
   kGenMM : generated matmul kernel
   kMM_NT : worst no-copy kernel
   kMM_TN : best no-copy kernel
   BIG_MM : large GEMM timing (usually N=1600); estimate of asymptotic peak
   kMV_N  : NoTranspose matvec kernel
   kMV_T  : Transpose matvec kernel
   kGER   : GER (rank-1 update) kernel
Kernel routines are not called by the user directly, and their
performance is often somewhat different than the total
algorithm (eg, dGER perf may differ from dkGER)


Reference clock rate=2394Mhz, new rate=2394Mhz
   Refrenc : % of clock rate achieved by reference install
   Present : % of clock rate achieved by present ATLAS install
```

	single precision				double precision			
	real		complex		real		complex	
Benchmark	Refrenc	Present	Refrenc	Present	Refrenc	Present	Refrenc	Present
kSelMM	535.0	551.4	525.4	509.6	311.5	312.7	298.0	296.5
kGenMM	175.5	174.0	175.5	173.6	160.5	159.7	165.4	166.9
kMM_NT	145.2	143.7	149.3	150.7	135.3	131.0	132.3	134.3
kMM_TN	163.2	158.0	161.1	164.6	148.7	144.8	146.0	155.4
BIG_MM	510.1	544.5	504.0	545.9	307.7	301.5	293.0	304.9
kMV_N	113.5	109.1	216.9	208.3	58.9	56.2	97.4	88.8
kMV_T	89.9	85.9	94.6	96.4	47.2	44.4	74.1	77.1
kGER	154.2	154.1	119.4	116.9	29.1	26.0	46.8	45.6

Figure 1: Normal results for `make time` on Core2Duo64SSE3

```
Reference clock rate=2200Mhz, new rate=1597Mhz
....
```

	single precision				double precision			
	real		complex		real		complex	
Benchmark	Refrenc	Present	Refrenc	Present	Refrenc	Present	Refrenc	Present
kSelMM	335.5	338.8	329.4	331.6	178.9	180.8	180.3	178.7
kGenMM	175.4	100.4	174.2	100.3	163.7	92.6	141.4	94.9
kMM_NT	142.0	86.8	141.2	92.0	125.3	85.2	138.1	88.8
kMM_TN	143.0	92.7	141.1	95.2	139.4	87.8	137.4	90.1
BIG_MM	327.1	325.2	318.6	320.0	169.8	171.3	171.0	172.0
kMV_N	61.4	35.5	139.3	98.9	47.2	30.7	71.9	74.2
kMV_T	73.6	53.6	75.3	62.5	31.6	20.2	52.7	36.6
kGER	43.6	28.8	91.8	65.1	23.7	18.3	46.8	40.3

Figure 2: Timings results when architectural defaults are compiled with substandard gcc4.1, rather than gcc4.2

has markedly depressed performance across the board. Anytime you see a pattern such as this, the first thing you should check is if you are using a recommended compiler, and if not, install and use that compiler.

On the other hand, if only your `BIG_MM` column is depressed, it is likely you have a bad setting for the `CacheEdge` or the complex-to-real crossover point (if the performance is depressed only for both complex types).

6.1 Contrasting non-default install performance

If you do not install using the architectural defaults, `make time` will only print out the `Present` columns. This gives you a good summary of ATLAS's library performance, but it can be hard to tell what is good and bad if you are not familiar with ATLAS on this hardware. Sometimes, ATLAS has architectural defaults for your platform, but your install doesn't use them. This is usually because the installer has specified the use of a non-default compiler, or has explicitly asked that the architectural defaults not be used, or has overridden the detection of the architecture, etc. In this case, `make time` does not do the comparison against the architectural defaults, and so only the `Present` columns are printed.

However, if you wish to ensure that your library is as good as one that uses the architectural defaults, then you can manually tell the program called by `make time` (`xatlbench` to do the comparison. The most common example would be you have switched to an unsupported compiler (eg., the Intel compiler), and now you want to see if the library you built using it is as fast or faster than the one using the default `gcc 4.2` compiler. Another example would be that you want to compare the performance of two closely related architectures. This is what we will do here, where we contrast the performance of the 32 and 64 bit versions of the library on my Core2Duo.

In order to manually do a comparison between a present install and any of the results stored in ATLAS's architectural defaults you'll need to perform the following steps:

1. `make time` issued in the `BLDdir` of your non-default install. This does the timings of the present build, and stores the results in `BLDdir/bin/INSTALL_LOG`.
2. `cd SRCdir/CONFIG/ARCHS`, and find the tarfile containing the results you wish to compare against. In our case, we choose `Core2Duo32SSE3.tgz` to compare against our own Core2Duo64SSE results.
3. `gunzip -c Core2Duo32SSE3.tgz | tar xvf -` untars the selected architectural results (replace `Core2Duo32SSE3.tgz` with the tarfile you have selected in step#2).
4. `cd BLDdir`
5. `./xatlbench -dp SRCdir/CONFIG/ARCHS/<ARCH> -dc BLDdir/bin/INSTALL_LOG`
   `xatlbench` is the program that compares two sets of results, with the `-dp` pointing to the previous (`Refrenc`) install result directory and `-dc` pointing to the current (`Present`) install result directory.

Figure 3 shows me doing this on my Core2Duo, with `SRCdir` = `/home/whaley/TEST/ATLAS3.7.36.0` and `BLDdir` = `/home/whaley/TEST/ATLAS3.7.36.0/obj64`, where we compare the present 64-bit install to the stored 32-bit install. We see that the 64-bit install, which gets to use 16 rather than 8 registers, is slightly faster for almost all kernels and precisions, as one might expect.

```
core2.home.net. cd /home/whaley/TEST/ATLAS3.7.36.0/obj64
core2.home.net. make time
..... lots of output .....
core2.home.net. pushd ~/TEST/ATLAS3.7.36.0/CONFIG/ARCHS/
core2.home.net. ls
BOZOL1.tgz          CreateTar.sh      MIPSICE964.tgz  POWER564.tgz
Core2Duo32SSE3/     HAMMER64SSE2.tgz  MIPSR1xK64.tgz  PPCG532AltiVec.tgz
Core2Duo32SSE3.tgz  HAMMER64SSE3.tgz  negflt.c        PPCG564AltiVec.tgz
Core2Duo64SSE3/     IA64Itan264.tgz   P432SSE2.tgz    USIV32.tgz
Core2Duo64SSE3.tgz  KillDirs.sh       P4E32SSE3.tgz   USIV64.tgz
CoreDuo32SSE3.tgz   Make.ext          P4E64SSE3.tgz
CreateDef.sh        Makefile          POWER432.tgz
CreateDirs.sh       MIPSICE932.tgz    POWER464.tgz
core2.home.net. gunzip -c Core2Duo32SSE3.tgz | tar xvf -
..... lots of output .....
core2.home.net. pushd
core2.home.net. ./xatlbench \
   -dp /home/whaley/TEST/ATLAS3.7.36.0/CONFIG/ARCHS/Core2Duo32SSE3 \
   -dc /home/whaley/TEST/ATLAS3.7.36.0/obj64/bin/INSTALL_LOG/
.....
Reference clock rate=2394Mhz, new rate=2394Mhz
.....
              single precision                 double precision
            ******************************   ******************************
                 real           complex           real           complex
            ---------------  ---------------  ---------------  ---------------
Benchmark   Refrenc Present  Refrenc Present  Refrenc Present  Refrenc Present
=========   ======= =======  ======= =======  ======= =======  ======= =======
  kSelMM      539.0   551.4    496.5   509.6    299.4   312.7    289.0   296.5
  kGenMM      165.1   174.0    165.1   173.6    156.1   159.7    153.8   166.9
  kMM_NT      137.6   143.7    134.7   150.7    115.7   131.0    123.5   134.3
  kMM_TN      116.3   158.0    112.3   164.6    101.3   144.8    110.9   155.4
  BIG_MM      521.3   544.5    476.5   545.9    282.6   301.5    282.8   304.9
   kMV_N       69.0   109.1    206.9   208.3     56.3    56.2     69.4    88.8
   kMV_T       84.8    85.9    117.3    96.4     48.0    44.4     87.9    77.1
    kGER       90.1   154.1    114.2   116.9     27.9    26.0     41.5    45.6
```

Figure 3: Comparing 32 and 64 bit libraries on a 2.4 Ghz Core2Duo

6.2 Discussion of timing targets

Presently, ATLAS times mostly kernel routines, which are used to build higher level routines that then appear in the BLAS or LAPACK. `kSelMM` is the matrix multiply kernel that is being used for large GEMM calls, which will be the best kernel found in the generator and multiple implementation searches. Therefore this kernel may be written in assembly on some platforms. `kGenMM` is the fastest generated kernel that matches `kSelMM`, and it may be used for some types of cleanup. All generated kernels are written in ANSI C, and thus their peak performance will strongly depend on the compiler being used.

`kMM_NT` and `kMM_TN` are two of the four generated kernels that will be used for small-case GEMM when we cannot afford to copy the input matrices. The last two characters indicate the transpose settings. The other two kernels' performance lies between these extremes: `NT` is typically the slowest kernel (all non-contiguous access), and `TN` is typically the fastest (all contiguous access).

`BIG_MM` is the only non-kernel timing we presently report, and it is the speed found when doing a large GEMM call. "Large" can vary by platform: it is typically $M = N = K = 1600$, except where we were unable to allocate that much memory, where it will be less. On many machines, this line gives you a rough asymptotic bound on BLAS performance.

The next three lines report Level 2 BLAS kernel performance (the Level 2 BLAS' performance will follow these kernels in roughly the same way that the Level 3 follow the GEMM kernels).

We should eventually supply an expanded timing comparison that would include higher level timings, such as LAPACK routines and threaded performance, but do not currently do so.

7 The ATLAS install step

This final optional step instructs ATLAS to copy the created libraries and include files into the appropriate directories, as specified in the configure step. This functionality is new, and so far is not bullet-proof (for instance, it copies only static libraries, and so presently fails to copy any dynamic libraries the user has built). From your `BLDdir`, it may be invoked by:

```
make install
```

By default, this command will copy all the static libraries to `/usr/local/atlas/lib` and all the user-includable header files to `/usr/local/atlas/include`. You may override this default directory during the configure step using the gnu-like flags `--prefix`, `--incdir` and/or `--libdir`. Assuming you didn't issue `--incdir` or `--libdir`, you can also override the prefix directory at install time with the command:

```
make install DESTDIR=<prefix directory to install atlas in>
```

8 Example: Installing ATLAS with full LAPACK on Linux/AMD64

In this section, I show a complete ATLAS install, including installing LAPACK. We assume I have already downloaded the tarfiles `atlas3.9.12.tar.bz2` and `lapack.tgz` into the `/home/whaley/dload` directory.

8.1 Figuring out configure flags

The system is a Fedora Core 8 system, which unfortunately uses the broken `gcc 4.1.2`, which would cripple ATLAS performance. Therefore, prior to installing ATLAS, I have installed gcc 4.2.1, with `--prefix=/home/whaley/local/gcc-4.2.1` I therefore add the following lines to my `.cshrc` so that ATLAS will use this gcc (it is put first in the path), and will be able to find the `gcc 4.2` libraries:

```
set path = (/home/whaley/local/gcc-4.2.1/bin $path)
setenv LD_LIBRARY_PATH /home/whaley/local/gcc-4.2.1/lib64:/home/whaley/local/gcc-4.2.1/lib
```

I source the C shell startup file, and then check that I'm now getting the correct compiler:

```
etl-opt8>source ~/.cshrc
etl-opt8>gcc -v
Using built-in specs.
Target: x86_64-unknown-linux-gnu
Configured with: ../configure --prefix=/home/whaley/local/gcc-4.2.1 --enable-languages=c
Thread model: posix
gcc version 4.2.1
```

Now, I don't need to pass a lot of flags to set what compiler to use, since ATLAS will find `gcc 4.2` as the first compiler, and it will have the libraries it needs to work. However, I want to build dynamic libraries for this install, so I know I'll need to add the `--shared` configure flag; config will automatically add the required `-fPIC` flag to all gnu compilers so they can build shared object code.

Now, I do a top on `etl-opt8` (the machine name) and see that I'm alone on the machine. Therefore, I will want to use the cycle-accurate x86-specific wall timer in order to improve the accuracy of my install. This requires me to figure out what the Mhz of my machine is. Under Linux, I can discover this with `cat /proc/cpuinfo`, which tells me `cpu MHz : 2100.000`. Therefore, I will throw `-D c -DPentiumCPS=2100`.

I want ATLAS to install the resulting libraries and header files in the directory `/home/whaley/local/atlas`, so I'll pass `--prefix=/home/whaley/local/atlas` as well.

I want a 64 bit install, and to build a full LAPACK library, so I will also want to throw `-b 64` and `--with-netlib-lapack-tarfile=/home/whaley/dload/lapack.tgz`.

8.2 Creating `BLDdir` and installing ATLAS

I'm ready to install ATLAS and LAPACK. I just need to untar the ATLAS tarfile, issue, create my `BLDdir`, and issue the previously selected flags to configure:

```
etl-opt8>bunzip2 -c ~/dload/atlas3.9.12.tar.bz2 | tar xfm -
etl-opt8>mv ATLAS ATLAS3.9.12.1
etl-opt8>cd ATLAS3.9.12.1/
etl-opt8>mkdir obj64
etl-opt8>cd obj64/
etl-opt8>../configure -b 64 -D c -DPentiumCPS=2100 -Fa alg --shared \
  --prefix=/home/whaley/local/atlas \
  --with-netlib-lapack-tarfile=/home/whaley/dload/lapack.tgz
................................................
............<A WHOLE LOT OF OUTPUT>.............
................................................

etl-opt8>ls
```

```
ARCHS/       Makefile          xconfig*   xprobe_3dnow*      xprobe_OS*
atlcomp.txt  Make.inc          xctest*    xprobe_arch*       xprobe_pmake*
atlconf.txt  Make.top          xf2cint*   xprobe_asm*        xprobe_sse1*
bin/         src/              xf2cname*  xprobe_comp*       xprobe_sse2*
include/     tune/             xf2cstr*   xprobe_f2c*        xprobe_sse3*
interfaces/  xarchinfo_linux*  xf77test*  xprobe_gas_x8632*  xprobe_vec*
lib/         xarchinfo_x86*    xflibchk*  xprobe_gas_x8664*  xspew*

etl-opt8>make
.....................................................
............<A WHOLE WHOLE LOT OF OUTPUT>.............
.....................................................
ATLAS install complete.  Examine
ATLAS/bin/<arch>/INSTALL_LOG/SUMMARY.LOG for details.
make[1]: Leaving directory '/home/whaley/TEST/ATLAS3.9.12.1/obj64'
make clean
make[1]: Entering directory '/home/whaley/TEST/ATLAS3.9.12.1/obj64'
rm -f *.o x* config?.out *core*
make[1]: Leaving directory '/home/whaley/TEST/ATLAS3.9.12.1/obj64'
1628.011u 153.212s 23:05.34 128.5%      0+0k 32+3325928io 0pf+0w
```

OK, in a little over 20 minutes, we've got ATLAS and LAPACK built. Now, we need to see if it passes the sanity tests, which we do by:

```
etl-opt8>make check
.....................................................
............<A WHOLE LOT OF COMPILATION>.............
.....................................................
DONE BUILDING TESTERS, RUNNING:
SCOPING FOR FAILURES IN BIN TESTS:
fgrep -e fault -e FAULT -e error -e ERROR -e fail -e FAIL \
               bin/sanity.out
8 cases: 8 passed, 0 skipped, 0 failed
4 cases: 4 passed, 0 skipped, 0 failed
8 cases: 8 passed, 0 skipped, 0 failed
4 cases: 4 passed, 0 skipped, 0 failed
8 cases: 8 passed, 0 skipped, 0 failed
4 cases: 4 passed, 0 skipped, 0 failed
8 cases: 8 passed, 0 skipped, 0 failed
4 cases: 4 passed, 0 skipped, 0 failed
DONE
SCOPING FOR FAILURES IN CBLAS TESTS:
fgrep -e fault -e FAULT -e error -e ERROR -e fail -e FAIL \
               interfaces/blas/C/testing/sanity.out | \
               fgrep -v PASSED
make[1]: [sanity_test] Error 1 (ignored)
DONE
SCOPING FOR FAILURES IN F77BLAS TESTS:
fgrep -e fault -e FAULT -e error -e ERROR -e fail -e FAIL \
               interfaces/blas/F77/testing/sanity.out | \
               fgrep -v PASSED
make[1]: [sanity_test] Error 1 (ignored)
DONE
make[1]: Leaving directory '/home/whaley/TEST/ATLAS3.9.12.1/obj64'
61.684u 6.485s 1:08.66 99.2%    0+0k 0+163768io 0pf+0w
```

So, since we see no failures, we passed. I get essentially the same output when I check the parallel interfaces (my machine has eight processors) via `make ptcheck`.

Now, I am ready to make sure my libraries are getting the expected performance, so I do:

```
etl-opt8>make time
.....................................................
.....<A WHOLE LOT OF COMPILATION>.....
.....................................................
                    single precision                     double precision
             ********************************   ********************************
                   real           complex           real           complex
             ---------------  ---------------  ---------------  ---------------
Benchmark    Refrenc Present  Refrenc Present  Refrenc Present  Refrenc Present
=========    ======= =======  ======= =======  ======= =======  ======= =======
  kSelMM       643.4   642.9    622.0   621.8    323.8   343.5    320.5   316.9
  kGenMM       191.4   192.1    161.8   174.1    178.3   164.3    172.9   172.4
  kMM_NT       140.0   138.5    127.4   129.3    137.4   136.1    126.4   131.8
  kMM_TN       165.2   165.3    159.8   157.0    163.0   161.6    158.0   155.2
  BIG_MM       604.1   617.0    601.8   599.8    311.3   332.3    309.2   292.1
   kMV_N        74.3    70.2    211.2   197.5     51.9    48.4    107.3    99.7
   kMV_T        82.2    79.8     97.2    95.3     46.4    43.9     77.6    73.3
    kGER        60.1    56.9    153.5   130.3     38.8    32.0     77.5    64.8
```

We see that load and timer issues have made it so there is not an exact match, but that neither install is worse overall, and so this install looks good! Now we are finally ready to install the libraries. We can do so, and then check what got installed by:

```
etl-opt8>make install
.............................................
.............<A LOT OF OUTPUT>.............
.............................................
etl-opt8>cd ~/local/atlas/
etl-opt8>ls
include/  lib/

etl-opt8>ls include/
atlas/  cblas.h  clapack.h

etl-opt8>ls include/atlas/
atlas_buildinfo.h      atlas_dr1kernels.h     atlas_strsmXover.h
atlas_cacheedge.h      atlas_dr1_L1.h         atlas_tcacheedge.h
atlas_cGetNB_gelqf.h   atlas_dr1_L2.h         atlas_trsmNB.h
atlas_cGetNB_geqlf.h   atlas_dsyr2.h          atlas_type.h
atlas_cGetNB_geqrf.h   atlas_dsyr.h           atlas_zdNKB.h
atlas_cGetNB_gerqf.h   atlas_dsyr_L1.h        atlas_zGetNB_gelqf.h
atlas_cmv.h            atlas_dsyr_L2.h        atlas_zGetNB_geqlf.h
atlas_cmvN.h           atlas_dsysinfo.h       atlas_zGetNB_geqrf.h
atlas_cmvS.h           atlas_dtGetNB_gelqf.h  atlas_zGetNB_gerqf.h
atlas_cmvT.h           atlas_dtGetNB_geqlf.h  atlas_zmv.h
atlas_cNCmm.h          atlas_dtGetNB_geqrf.h  atlas_zmvN.h
atlas_cr1.h            atlas_dtGetNB_gerqf.h  atlas_zmvS.h
atlas_cr1kernels.h     atlas_dtrsmXover.h     atlas_zmvT.h
atlas_cr1_L1.h         atlas_pthreads.h       atlas_zNCmm.h
atlas_cr1_L2.h         atlas_sGetNB_gelqf.h   atlas_zr1.h
atlas_csNKB.h          atlas_sGetNB_geqlf.h   atlas_zr1kernels.h
atlas_csyr2.h          atlas_sGetNB_geqrf.h   atlas_zr1_L1.h
atlas_csyr.h           atlas_sGetNB_gerqf.h   atlas_zr1_L2.h
atlas_csyr_L1.h        atlas_smv.h            atlas_zsyr2.h
atlas_csyr_L2.h        atlas_smvN.h           atlas_zsyr.h
```

```
atlas_csysinfo.h       atlas_smvS.h           atlas_zsyr_L1.h
atlas_ctGetNB_gelqf.h  atlas_smvT.h           atlas_zsyr_L2.h
atlas_ctGetNB_geqlf.h  atlas_sNCmm.h          atlas_zsysinfo.h
atlas_ctGetNB_geqrf.h  atlas_sr1.h            atlas_ztGetNB_gelqf.h
atlas_ctGetNB_gerqf.h  atlas_sr1kernels.h     atlas_ztGetNB_geqlf.h
atlas_ctrsmXover.h     atlas_sr1_L1.h         atlas_ztGetNB_geqrf.h
atlas_dGetNB_gelqf.h   atlas_sr1_L2.h         atlas_ztGetNB_gerqf.h
atlas_dGetNB_geqlf.h   atlas_ssyr2.h          atlas_ztrsmXover.h
atlas_dGetNB_geqrf.h   atlas_ssyr.h           cmm.h
atlas_dGetNB_gerqf.h   atlas_ssyr_L1.h        cXover.h
atlas_dmv.h            atlas_ssyr_L2.h        dmm.h
atlas_dmvN.h           atlas_ssysinfo.h       dXover.h
atlas_dmvS.h           atlas_stGetNB_gelqf.h  smm.h
atlas_dmvT.h           atlas_stGetNB_geqlf.h  sXover.h
atlas_dNCmm.h          atlas_stGetNB_geqrf.h  zmm.h
atlas_dr1.h            atlas_stGetNB_gerqf.h  zXover.h

etl-opt8>ls lib/
libatlas.a   libcblas.so    liblapack.a   libptcblas.so
libatlas.so  libf77blas.a   liblapack.so  libptf77blas.a
libcblas.a   libf77blas.so  libptcblas.a  libptf77blas.so
```

The shared object support in ATLAS is still experimental, so we can get some idea if our shared objects work by running an undocumented tester. To try a dynamically linked LU factorization, we:

```
animal>cd ../bin
animal>make xdlutst_dyn
............................................................
............<A WHOLE LOT OF UP-TO-DATE CHECKING>.............
............................................................
make[1]: Leaving directory '/home/whaley/numerics/ATLAS3.7.38/animal64/bin'
gfortran -O -fPIC -m64 -o xdlutst_dyn dlutst.o \
   /home/whaley/numerics/ATLAS3.7.38/animal64/lib/libtstatlas.a \
   /home/whaley/numerics/ATLAS3.7.38/animal64/lib/liblapack.so \
   /home/whaley/numerics/ATLAS3.7.38/animal64/lib/libf77blas.so \
   /home/whaley/numerics/ATLAS3.7.38/animal64/lib/libcblas.so \
   /home/whaley/numerics/ATLAS3.7.38/animal64/lib/libatlas.so \
   -Wl,--rpath /home/whaley/numerics/ATLAS3.7.38/animal64/lib

animal>./xdlutst_dyn
NREPS  Major      M      N    lda  NPVTS      TIME     MFLOP     RESID
=====  =====  =====  =====  =====  =====  ========  ========  ========
    0  Col      100    100    100     95     0.001  1273.153 1.416e-02
    0  Col      200    200    200    194     0.002  2453.930 1.087e-02
    0  Col      300    300    300    295     0.007  2574.077 8.561e-03
    0  Col      400    400    400    394     0.017  2531.312 8.480e-03
    0  Col      500    500    500    490     0.031  2701.090 7.610e-03
    0  Col      600    600    600    594     0.051  2796.150 8.332e-03
    0  Col      700    700    700    693     0.081  2832.877 7.681e-03
    0  Col      800    800    800    793     0.116  2938.840 7.091e-03
    0  Col      900    900    900    893     0.161  3014.142 6.856e-03
    0  Col     1000   1000   1000    995     0.221  3019.330 7.097e-03

10 cases ran, 10 cases passed
```

So, we appear to be good, and the install is complete! Now we point our users to the installed libs, and wait for the error reports to roll in.

9 Special Instructions for some platforms

9.1 Special Instructions for Windows users

ATLAS presently requires cygwin in order to install under Windows. Cygwin provides a Unix-style shell environment (including standard utilities such as `gcc` and `make`) for Windows. Cygwin is free, and can be downloaded from `www.cygwin.com`. We presently do not support Interix (AKA Windows Services for Unix, etc.) as provided by Microsoft, but a user has submitted code to help with this, and so we hope to add support in the future. We have had requests to support MinGW (`http://www.mingw.org/`), but no one has submitted suggested code to help, and I have never successfully figured out how to install and use it, so this is probably not coming soon unless something changes.

Once cygwin is installed, you are ready to install ATLAS. If you want to call ATLAS from code using `gcc` and `gfortran`, then you can just install as usual.

If you want to call ATLAS from code compiled by native compilers such as the Intel or Microsoft compilers, you must set up some environment variables so that these compilers can be called from cygwin's shell. Details on how do do this are available in the ATLAS errata file:

`http://math-atlas.sourceforge.net/errata.html#WinComp`

If you want multithreaded (eg., shared-memory parallel) ATLAS libraries, you must use `gcc` to compile the main library, and if you use a native compiler for interface compilation, manually link to the cygwin library. This is because ATLAS uses the POSIX threading standard, which of course Microsoft does not support, and so you need the cygwin emulation layer to use a decade-old standard.

9.2 Special instructions for AIX

Under AIX, it is critical that you define an envirnment variable indicating whether you are building 64 or 32 bit libraries, and this definition must match what you pass to `configure` via the `-b` flag. You need to define the environment variable `OBJECT_MODE` to either 64 or 32, depending on which of these you pass to `configure` using the `-b` flag. So, if you are building 64-bit libraries and you use a `bash` derivative shell, you would issue `export OBJECT_MODE=64` before starting the ATLAS configure step. On the other hand, if you use a `csh` derivative shell and want to build 32 but libaries, you would need to issue `setenv OBJECT_MODE 32` before the build step.

9.3 Special instructions for SunOS

Solaris has its own version of the Unix utilities, which differ sharply from the more common gnu tools. In particular, SunOS offers two fgreps, one of which works correctly for ATLAS's `make check` step, and one of which does not. On my SunOS machine, I had to make sure `/usr/xpg4/bin` was in my path before `/bin` in order to get an `fgrep` that can take multiple expression arguments (as `make check` requires).

Also, if gcc isn't compiled with with the correct gnu utilities, ATLAS may fail to autodetect the assembly dialect of your machine. This will cause the build to fail since it can't assemble the UltraSPARC assembly kernels, and you can see if it happened by examining your `Make.inc`'s ARCHDEF macro. If this macro does not include the definition

`-DATL_GAS_SPARC`, then this has happened to you. On some systems, you can get the install to work by adding the flag `-s 3` to your `configure` invocation. If this still doesn't fix the problem, you'll need to get a better gcc install. Note that this error causes linking to assembled files to die with messages like:

```
ld: fatal: relocation error: R_SPARC_32: file /var/tmp//ccccPppx.o:
    symbol <unknown>: offset 0xff061776 is non-aligned
```

10 Troubleshooting

The first thing you need to do is scope the errata file to see if your problem is already covered:

`http://math-atlas.sourceforge.net/errata.html`

Probably the most common error is when ATLAS dies because its timings are varying widely. This can often be fixed with a simple restart, as described:

`http://math-atlas.sourceforge.net/errata.html#tol`

If you are unable to find anything relevant in the errata file, you can submit a support request to the ATLAS support tracker (**not** the bug tracker, which is for developer-confirmed bugs only):

`https://sourceforge.net/tracker/?atid=379483&group_id=23725&func=browse`

When you create the support request, be sure to attach the error report. It should appear as `BLDdir/error_<arch>.tgz`. If this file doesn't exist, you can create it by typing `make error_report` in your `BLDdir`. More details on submitting support requests can be found in the ATLAS FAQ at:

`http://math-atlas.sourceforge.net/faq.html#help`

References

[1] Bjarne S. Andersen, Fred G. Gustavson, and Jerzy Wasniewski. A recursive formulation of cholesky factorization of a matrix in packed storage. Technical Report UT CS-00-448, LAPACK Working Note No.146, University of Tennessee, 2000.

[2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, PA, 3rd edition, 1999.

[3] L. Susan Blackford, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, Michael Heroux, Linda Kaufman Andrew Lumsdaine, Antoine Petitet, Roldan Pozo, Karin Remington, and R. Clint Whaley. An Updated Set of Basic Linear Algebra Subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.

[4] S. Blackford, G. Corliss, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, C. Hu, W. Kahan, L. Kaufman, B. Kearfott, F. Krogh, X. Li, Z. Maany, A. Petitet, R. Pozo, K. Remington, W. Walster, C. Whaley, J. Wolff, and V. Gudenberg. Document for the Basic Linear Algebra Subprograms (BLAS) Standard: BLAS Technical Forum. `http://www.netlib.org/cgi-bin/checkout/blast/blast.pl`, 1999.

[5] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.

[6] J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. Algorithm 656: An extended Set of Basic Linear Algebra Subprograms: Model Implementation and Test Programs. *ACM Transactions on Mathematical Software*, 14(1):18–32, 1988.

[7] J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. An Extended Set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, 1988.

[8] E. Elmroth and F. Gustavson. Applying recursion to serial and parallel qr factorization leads to better performance. *IBM Journal of Research and Development*, 44(4):605–624, 2000.

[9] F. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development*, 41(6):737–755, 1997.

[10] F. Gustavson, A. Henriksson, I. Jonsson, B. Kågström, and P. Ling. Recursive blocked data formats and blas's for dense linear algebra algorithms. In B. Kågström, J. Dongarra, E. Elmroth, and J. Waśniewski, editors, *Applied Parallel Computing, PARA'98*, Lecture Notes in Computer Science, No. 1541, pages 195–206, 1998.

[11] R. Hanson, F. Krogh, and C. Lawson. A Proposal for Standard Linear Algebra Subprograms. *ACM SIGNUM Newsl.*, 8(16), 1973.

[12] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, 1979.

[13] S. Toledo. Locality of reference in lu decomposition with partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 18(4), 1997.

[14] R. Clint Whaley and Jack Dongarra. Automatically Tuned Linear Algebra Software. Technical Report UT-CS-97-366, University of Tennessee, December 1997. `http://www.netlib.org/lapack/lawns/lawn131.ps`.

[15] R. Clint Whaley and Jack Dongarra. Automatically tuned linear algebra software. In *SuperComputing 1998: High Performance Networking and Computing*, 1998. CD-ROM Proceedings. **Winner, best paper in the systems category.** `http://www.cs.utsa.edu/~whaley/papers/atlas_sc98.ps`.

[16] R. Clint Whaley and Jack Dongarra. Automatically Tuned Linear Algebra Software. In *Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999. CD-ROM Proceedings.

[17] R. Clint Whaley and Antoine Petitet. Atlas homepage. `http://math-atlas.sourceforge.net/`.

[18] R. Clint Whaley and Antoine Petitet. Lapack homepage. `http://www.netlib.org/lapack/`.

[19] R. Clint Whaley and Antoine Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2):101–121, February 2005. `http://www.cs.utsa.edu/~whaley/papers/spercw04.ps`.

[20] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.

[21] R. Clint Whaley and Peter Soendergaard. A collaborative guide to atlas development. `http://math-atlas.sourceforge.net/devel/atlas_devel/`.