ATLAS Installation Guide *

R. Clint Whaley †

July 27, 2016

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 [20, 17, 18, 19, 23, 22]. ATLAS (Automatically Tuned Linear Algebra Software), is an empirical tuning system that produces a BLAS [7, 8, 9, 13, 14] (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 FORTRAN package for solving the most commonly occurring problems in numerical linear algebra. LAPACK is available as an open source FORTRAN package from netlib [21], 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 most of the routines that involve the LU, QR and Cholesky factorizations. ATLAS's implementation uses pure recursive version of LU and Cholesky based on the work of [15, 11, 12, 1], and the QR version uses the hybrid algorithm with static outer blocking and panel recursion described in [10]; the static blocking is empirically tuned as described in [16]. In parallel, these routines are further sped up by the PCA panel factorization [6] and the threading techniques discussed in [5]. The standard LAPACK routines use statically blocked routines, which typically run slower than recursively blocked for all problem sizes.

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. The netlib LAPACK has recently begun supporting Intel's propriatary C interface, which is incompatible with the C BLAS as well as ATLAS's C interface, as well as performaing a host of unnecessary matrix transpositions. 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/atlas3.10.3.tar.bz2 | tar xfm -
mv ATLAS ATLAS3.10.3
```

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

If ATLAS's configure detects that CPU throttling is enabled, it will kill itself. The reason is that ATLAS depends on timings to tune the library to your machine. OS-controlled throttling is so course-grained that essentially all timings become random. What this means is that instead of optimizing the code, ATLAS winds up randomly transforming the code, sometimes resulting in better code, and just as often, making the code slower.

If throttling is enabled, the only way to get a decent ATLAS install is therefore to turn it off. Most machines can switch it off in the BIOS, but on machines with newer Linux kernels even this does not stop pstate throttling. To tell linux not to ignore your bios settings you must log in as root/sudo, and issue something similar to:

1. Edit `/etc/default/grub`
2. Find line with: `GRUB_CMDLINE_LINUX_DEFAULT`
3. Append the following to args: `intel_pstate=disable`
4. Update grub: `update-grub`
5. Reboot

For modern Linuxes, you can turn off throttling using `cpufreq-set`. Some systems allow you to set the speed of individual cores, and others force groups of cores to run at same speed. Therefore, on a 4-core system, your frequency settings may not take hold until you've set all 4-cores to the desired setting. For instance:

```
/usr/bin/cpufreq-set -g performance -c 0
/usr/bin/cpufreq-set -g performance -c 1
/usr/bin/cpufreq-set -g performance -c 2
/usr/bin/cpufreq-set -g performance -c 3
```

Under MacOS or Windows, you may be able to change this under the power settings. I have reports that issuing "powercfg /q" in cmd.exe under windows will tell you whether windows is throttling or not.

Note that if you worry about the system overheating, you should be able to get decent (not perfect) tuning by forcing any constant clock rate (rather than forcing the maximum rate, as above).

If you cannot turn off cpu throttling you must pray that ATLAS already has architectural defaults. If so, ATLAS will likely still have poor performance, but at least not all code transformation decisions will be taken at random. To proceed with a random ATLAS tuning in the face of CPU throttling, you can throw the the configure flag: `--cripple-atlas-performance`.

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 3.8.1), and fine-tune the timer used, as discussed in Section 3.5. 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 atlas3.10.x.tar.bz2 | tar xfm -    # create SRCdir
mv ATLAS ATLAS3.10.x                          # get unique dir name
cd ATLAS3.10.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
  --with-netlib-lapack-tarfile=/home/whaley/dload/lapack-3.4.1.tgz
make build                                    # tune & build lib
make check                                    # sanity check correct answer
make ptcheck                                  # sanity check parallel
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 (§3.4), 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 (§3.5). Finally, we consider a few miscellaneous flags (§3.8), including the flag telling ATLAS whether the resulting libraries should assume a 64 or 32 bit address space (§3.8.1).

[1] Optional step

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-3.4.1.tgz`, you would add the following to your `configure` flags:

```
--with-netlib-lapack-tarfile=/home/whaley/dload/lapack-3.4.1.tgz
```

Configure then auto-builds a `make.inc` for LAPACK to use, and builds netlib LAPACK as part of the ATLAS install process. ATLAS 3.10.0 was tested tow work with LAPACK v3.4.1 and 3.3.1.

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, QR 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.

3.1.2 Obtaining netlib's LAPACK

You can download the LAPACK reference implementation from `www.netlib.org/lapack/`.

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`. 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! For the open source version of `clang`, gcc was always faster on all machines we had access to. Apple's proprietary clang seems to be a good deal faster, so OS X user's may want to try a search with both `clang` and `gcc` to see which is the fastest if both are available. See Section 9.3 for how to force clang to be used. Intel's `icc` was not tried for this release since it is a non-free compiler, but 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.

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. Under Unix, most compilers interoperate with the GNU compilers, and so you can build ATLAS with the GNU compilers, and then simply link to the resulting libs with the compiler of your choice.

On Windows, if you want to build ATLAS for linking with native libraries such as MSVC++, then you can build ATLAS with the MinGW compilers, which are GNU compilers that are made to natively interoprate with native Windows compilers. See Section 9.1 for more information.

For those who insist on monkeying with other compilers, Section 3.2.2 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>'
```

For 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 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`.

If you use the `-C` flag, then you are overriding ATLAS's compiler choice (based on the abbrevation you specify, as described below), `-F` means to override the flags for that compiler, and `-Fa` tells configure that you want to keep ATLAS's default flags, but wish to append your own list of flags to them.

All of these flags take an abbreviation (`<abbr>`) describing the particular compiler/flag to override/append, where `<abbr>` is one of,

- One of the already discussed compiler abbreviations (eg, `xc`, `gc`, `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).

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 3.8.4.

3.2.3 Forcing ATLAS to use a particular gcc

ATLAS contains architectural defaults allowing installers to skip most of the empirical tuning, with different platforms using different gcc versions. To see what gcc version was used to build archdefs for your system, you can scope `GetBestGccVers` in `ATLAS/CONFIG/src/probe_comp.c`. By default, ATLAS will search for this version on your system during configure, and if it can't find it, it will select the closest version number that it can find. Not even later versions of the compiler are necessarily better to use, since both performance and correctness regressions are relatively common. However, many users wish to force ATLAS to use a particular gcc, even when they have many different gccs installed. The easiest way to force ATLAS to use a particular gcc for all C compilers is:

```
-C acg /full/path/to/your/gcc
```

If you specify only the name and not the path (eg., "`-C alg gcc-4.4`", then ATLAS will search for the named compiler in your PATH variable. The safest approach is to give the full path to the compiler if gcc choice is critical to you. If you want also specify the `gfortran` to use, additionally add the flag:

```
-C if /full/path/to/your/gfortran
```

IMPORTANT NOTE: If you use a different gcc than preferred version, 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 you can tolerate a long install time, you may also want to tell ATLAS not to use the architectural defaults, as described in Section 3.8.4.

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.

3.2.5 Improving flag selection using mmflagsearch

If you are on a architecture or using a gcc for which configure does not suggest flags, or if you believe the present set is out-of-date, you can quickly search through a host of compiler flags to find the best set for a given gemm kernel using the specialized routine mmflagsearch.c. To do this, you need a working install, typically installed with your best guess at good flags. Now, in your `BLDdir/tune/blas/gemm` directory, issue `make xmmflagsearch`.

The idea behind this search is that it takes an ATLAS GEMM kernel description file (output from one of the ATLAS searches), and then tries a series of flags given in another file, and returns to you the best combination found. The important flags are:

`-p [s,d,c,z]` : set type/precision prefix

`-f <flagfile>` : file containing all flags to try

`-m <mmfile>` : mmsearch output file describing kernel to time

The `mmfile` is the matmul kernel that you wish to use to find the best flags, and if this argument is omitted the search will automatically read `res/<pre>gMMRES.sum`, which is the best kernel found for the during the prior install using the scalar ANSI C generator. If bad flags have caused this search to generate a weird file, you can copy this file to a new name, and then hand edit it to have the features you like.

In the `flagfile`, any line beginning with '`#`' is ignored. This file has a special format that is more easily understood once you understand the method of the search. The user provides one line for any flags that should always appear (examples include things like `-fPIC`, `-m64`, `-mcpu=XXX`, etc.). This is given on the first line.

Now, the way the search is going to work is that first it will find the appropriate optimization level and fundamental flag combination, which will be searched by trying all combinations of these flags. Once these baseline flags are determined, all remaining flags will be tried one after the other using a greedy linear search. With this in mind, the format of this file is:

```
Required flags for all cases (eg. -fPIC -m64 -msse3 -mfpmath=sse)
<N>      Number of optimization level lines
<lvlflagset1>
....
<lvlflagsetN>
<F>      Number of fundamental flag lines
<fundflagset1>
....
<fundflagsetF>
# Now list any number of modifier flag lines
flag set 1
flag set 2
...
flag set X
```

So, the way this search is going to work is that we will first try all $N \times (F + 1)$ combinations of the levels and fundamental flags, and choose a best-performing set. We will

then try adding every provided modifier flag line to the best found combination. The best performing list will be given.

To create such a flag file one usually scopes the compiler documentation, and finds all performance-oriented flags. For gcc, you can make mmflagsearch give you a template that includes all non-architecture-specific optimization flags (as found in the documentation for gcc 4.2) by running `./xmmflagsearch -f gcc`. This will create a file called `gccflags.txt` in the current directory, which presently has a format like:

```
REPLACE THIS LINE WT ARCH-DEP FLAGS ALWAYS USED (eg, -fPIC -m64 -msse3)
4
-O2
-O1
-O3
-Os
6
-fschedule-insns
-fno-schedule-insns
-fschedule-insns2
-fno-schedule-insns2
-fexpensive-optimizations
-fno-expensive-optimizations
# Flags to probe once optimization level is selected
...
whole boatload of flags
...
```

A similar file will be produced with some clang flags if you substitute `clang` for `gcc` above, though I was not able to find any central list of flags that I trusted, so the ones produced are probably insufficient and may not work on all systems.

Now lets see an example of this working on my ARM embedded machine. The first thing I do is replace the first line with my mandatory flags:

```
-mfpu=vfpv3 -mcpu=cortex-a8
```

I then add two architecture-specific flags to the auto-generated general flag list (might want to try a lot more, this is just an example), which in this case are:

```
-mtune=cortex-a8
-mno-thumb
```

An extract of this search is shown in Figure 1.

```
FINDING BEST FLAGS USING MATMUL KERNEL:
ID=0 ROUT='dgmm.c' AUTH='Whaley/emit_mm' TA='T' TB='N' \
   MULADD=1 PREF=1 LAT=5 NFTCH=2 IFTCH=6 FFTCH=1 KBMAX=0 KBMIN=0 KU=1 NU=5  \
   MU=4 MB=80 NB=80 KB=80 L14NB=0 PFBCOLS=0 PFABLK=0 PFACOLS=0 STFLOAT=0  \
   LDFLOAT=0 AOUTER=0 LDAB=1 BETAN1=0 LDISKB=1 KUISKB=0 KRUNTIME=0 NRUNTIME=0  \
   MRUNTIME=0 LDCTOP=0 X87=0  \
   MFLOP=5.848105e+02
FINDING BEST FLAG SETTINGS FOR THIS MATMUL KERNEL:
...Trying optlvls using base flags: '-mfpu=vfpv3 -mcpu=cortex-a8'
   1. mf=557.83, flags='-O2'
    ---> Opt level '-O2' is better!
   2. mf=558.18, flags='-O2 -fschedule-insns'
   3. mf=564.07, flags='-O2 -fno-schedule-insns'
    ---> Opt combo '-O2 -fno-schedule-insns' is better!
   4. mf=564.53, flags='-O2 -fno-schedule-insns -fschedule-insns2'
   5. mf=572.04, flags='-O2 -fno-schedule-insns -fno-schedule-insns2'
    ---> Opt combo '-O2 -fno-schedule-insns -fno-schedule-insns2' is better!
   6. mf=572.11, flags='-O2 -fno-schedule-insns -fno-schedule-insns2
                        -fexpensive-optimizations'
   7. mf=573.24, flags='-O2 -fno-schedule-insns -fno-schedule-insns2
                        -fno-expensive-optimizations'
   8. mf=572.34, flags='-O1'
   ... Bunch of cases elided ...
  27. mf=562.53, flags='-Os -fno-schedule-insns -fno-schedule-insns2
                        -fexpensive-optimizations'
  28. mf=563.81, flags='-Os -fno-schedule-insns -fno-schedule-insns2
                        -fno-expensive-optimizations'
...All cases using flags: '-O2 -mfpu=vfpv3 -mcpu=cortex-a8 -fno-schedule-insns
                           -fno-schedule-insns2'
  29. mf=571.41, flags='-mtune=cortex-a8'
  30. mf=572.32, flags='-mno-thumb'
  31. mf=571.02, flags='-fno-cprop-registers'
   ... Bunch of cases elided  ...
  42. mf=574.69, flags='-fomit-frame-pointer'
  43. mf=572.39, flags='-foptimize-register-move'
  44. mf=571.79, flags='-fno-optimize-register-move'
  45. mf=592.68, flags='-fprefetch-loop-arrays'
    ---> Adding flag '-fprefetch-loop-arrays'!
  46. mf=572.65, flags='-fno-prefetch-loop-arrays'
   ... Bunch of cases elided  ...
  90. mf=594.61, flags='-falign-loops=8'
  91. mf=594.65, flags='-falign-loops=16'
  92. mf=594.37, flags='-falign-loops=32'
BEST FLAGS GIVE MFLOP=592.68 (6.25% improvement over first case):
   '-O2 -mfpu=vfpv3 -mcpu=cortex-a8 -fno-schedule-insns -fno-schedule-insns2
    -fprefetch-loop-arrays'
```

Figure 1: Result of `./xmmflagsearch -p d -f gccflags.txt` on ARM

3.3 Handling hyperthreading, SMT, modules, and other horrors

Most modern machines include multiple virtual processors on each core; this basic idea is called hyperthreading by Intel, SMT by IBM, and other names by others. AMD took it to a new level with the dozer architecture, where two integer units share an FPU.

All of these techniques mainly help for codes that are extremely inefficient: by allowing multiple threads that cannot drive the architecture's backend at its maximal rate, you can get the backend running nearer to peak. However, for efficient codes that can drive the bottleneck backend functional units at their maximal rate, these strategies can cause slowdowns that range from slight to catastrophic, depending on the situation. For ATLAS, the main problem is usually that the increased contention on the caches caused by the extra threads tends to thrash the caches.

The only architecture where I have seen the use of these virtual processors yield speedups on most ATLAS operations is the Sun Niagara; I believe the machine I observed speedups on was a T2, but this might be true for any of the T-series.

I recommend that HPC users turn off these virtual processors on all other systems, which is usually done either in the BIOS or by OS calls. If you do not have root, or if you have less optimized applications that are getting speedup from these virtual cores, you can tell ATLAS to use only the real cores if you learn a little about your machine. Unfortunately, ATLAS cannot presently autodetect these features, but if you experiment you can discover which affinity IDs are the separate cores, and tell ATLAS to use only these cores. The general form is to add the following to your usual configure flags:

```
--force-tids="# <thread ID list>"
```

3.3.1 Handling modules on AMD systems

For instance, on my AMD Dozer system, there are 8 integer cores, but only 4 FPUs, and so for best performance we would like to use 4 threads rather than 8, and be sure to not use any integer core that shares an FPU. A little testing showed that on my system, core IDs 0, 1, 3, and 6 are all independent of each other, and so I can tell ATLAS to use only these four cores in threaded operations by adding this flag to configure:

```
--force-tids="4 0 1 3 6"
```

On my system, this actually slightly reduces parallel GEMM performance, but noticably improves factorization performance.

3.3.2 Handling SMT on IBM POWER

Similarly, an IBM Power7 I have access to has 8 physical cores, but offers 64 SMT units. If you install with the default flags, your parallel speedup for moderate sized DGEMMs is around 4.75. On the other hand, if you add:

```
--force-tids="8 0 8 16 24 32 40 48 56"
```

Then the parallel DGEMM speedup for moderate sized problems is more like 6.5.

I also have access to a POWER8 machine, with four physical cores, that are again shared 8-way, leading to the need to add to configure:

```
--force-tids="4 0 8 16 24"
```

3.4 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). As long as you are using gnu compilers, all you need to add to your `configure` command is:

```
--shared
```

For any non-gnu compiler, you will additionally have to tell configure what flags are needed to tell the compiler to produce a shared library-compatible object file (you can skip this step if the compiler does so by default.

ATLAS always builds the static libraries, but the `--shared` command adds an additional step to the install which also builds two shared libraries:

libsatlas.[so,dylib,dll]: This library contains all serial APIs (serial lapack, serial BLAS), and all ATLAS symbols needed to support them.

libtatlas.[so,dylib,dll]: This library contains all parallel APIs (parallel LAPACK and parallel BLAS) and all ATLAS symbols needed to support them.

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.

WINDOWS NOTE: If you are on Windows and using the MinGW compilers to work natively in windows (outside cygwin), then please see the errata file for additional instructions on enabling this porting.

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.

3.5 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 thus tends to produce relatively poorly optimized libraries. 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 kernel 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.

3.6 Explicitly overriding architecture detection

The ATLAS configure script may not recognize your architecture. If this happens, ATLAS will fall back on using generic flags, and a full search. To ge the best performance often takes two installs, where first you do a generic install, then you search for the best flags (see §3.2.5).

Sometimes, however, ATLAS actually has support for your architecture (or to a machine that is functionally equivalent), but it doesn't recognize it. In this case, you can explictly tell ATLAS which architecture it should assume using the `-A <name>` configure flag. To see a list of supported architectures, after any ATLAS configure step (even one with bad architecture and flags) you can do:

```
make xprint_enums
./xprint_enums
```

Note that there is not a one-to-one correspondence between ATLAS's names (chosen based on ATLAS-visable changes) and, for instance Intel code or marketing names. Table 1 shows the maping of ATLAS architectures to some of the more recent Intel code names.

ATLAS	Intel code name
Corei1	Nahalem / Westmere
Corei2	Ivy/Sandy bridge
Corei3	haswell/broadwell[-e]
Corei4	skylake

Table 1: ATLAS arch string & Intel code names

Therefore, if configure was failing to autodetect the architecture, but I knew the machine was a haswell, I could add the following to configure to get the haswell architectural defaults:

```
-A Corei3
```

3.6.1 Reporting configure failures

In general, if ATLAS does not recognize your architecture, I recommend submitting a support request (even if you make your own library that you are happy with) so that ATLAS can be extended to recognize the architecture for others.

To get support on a configure failure, you'll need to capture the result of the failing step, preferably invoking configure in verbose mode by adding `-v 2` to your configure flags. On most Unix boxes, you can capture the output by typing "`script`", then doing the configure, then typing "`exit`" to stop recoreding. This will create a file called "typescript" which you can upload to show what is happening to you.

You can also get the output of configure by using redirection, just be sure to redirect both `stdout` and `stderr`!

3.7 Building Generic x86 libraries

Many users ask how ATLAS can be used to build libraries that will run on all x86 platforms. In general, this is a bad idea: ATLAS gets its speed by specializing for particular platforms, so the more generic a library is the less performance it will achieve! Note that libraries like MKL can do well across many platforms by having fat binaries, where each kernel routine has actually been seperately tuned for many different platforms, and then queries something like CPUID to determine what sublibrary to call dynamically. ATLAS does not have the ability to build fat libraries.

So, users wanting generic x86 libraries will definitely lose performance in ATLAS, but many system admins have asked for this feature, and so I have added it to ATLAS. The idea is to get you libraries that get better performance than the reference BLAS, but whose percentage of peak may be woefully low, but that will run on a variety of platforms. You can do this by artificially overriding ATLAS's architecture detection, and manually telling configure to use use some generic architectural defaults that have been created, as described in the following paragraphs.

Never use these libraries unless this portability is absolutely required. They must use portable settings for blocking, for instance, which will mean that on many platforms they will use only a fraction of the actual cache, causing large performance drops. Even worse, peak performance may by reduced by as much as factor of 8, due to not using the proper

ISA extension. The most portable ISA uses only the x87 unit, which has a much lower peak rate on most modern machines (eg., an Intel sandy bridge can do 16 flops/cycle using AVX in single precision, but only 8 flops/cycle if using SSE, and only 2 flops/cycle using the x87 unit).

3.7.1 Building generic libraries for any x86 with an x87 FPU (PPRO & later)

To build binaries that will run on any x86 platform that implements an x87 FPU unit (I believe this is all Intel architectures from PentiumPRO onwards, and any AMD platform since at least the original Athlon), add the following flags to your configure line:

```
-b 32 -V -1 -A x86x87
```

`-b 32` ensures that you build the 32-bit libraries, which is necessary, since most older machines do not implement x86-64 (AMD64). The `-V -1` says use no ISA extensions beyond the original x86 spec. The `-A x86x87` selects an artificial architecture providing you with portable (but slow!) architectural defaults.

3.7.2 Building generic libraries for SSE1 (PIII & later)

If all of the machine you target implement the original SSE ISA extension, then you can improve your single precision peak by allowing ATLAS to use SSE1 kernels. These libraries should work on the Pentium III or any following Intel chip; for AMD it should work with the Athlon XP or any following chip.

To build this generic target, add the following flags to your configure line:

```
-b 32 -V 128 -A x86SSE1
```

`-b 32` ensures that you build the 32-bit libraries, which is necessary, since most older machines do not implement x86-64 (AMD64). The `-V 128` says that the original x86 ISA is extended by SSE1 only. The `-A x86SSE1` selects an artificial architecture providing you with portable (but slow!) architectural defaults.

3.7.3 Building generic libraries for SSE2 (P4 & later)

If all your target machines at least have SSE2, then you can use SSE for double precision computations as well. For intel, chips starting with the Pentium 4 had SSE2, and I believe AMD introduced SSE2 with the Opteron processor. Early P4's are 32-bit, and later are 64-bit, so you will need to decide yourself if you want `-b 32` or `-b 64` for your libraries. Right now, we have introduced architectural defaults only for 32-bits (meaning the 64-bit installs will take much longer). So, to limit the ISA to 32-bit, add these flags to your configure line:

```
-b 32 -V 192 -A x86SSE2
```

3.7.4 Selecting a good generic CacheEdge

ATLAS uses the CacheEdge macro set in `BLDdir/include/atlas_cacheedge.h` and `atlas_tcacheedge.h` to control the L2-cache blocking for the serial and threaded libraries, respectively. You'll want to be sure this value is either set to the minimum of the L2SIZE of any target architecture, or ridiculously large, so that no effective L2 blocking is done. So, if you are using

non-celeron x86, it almost always safe to set this value (in both files) to 256K (262144), since almost all archs have at least this much cache. If you know your target machines have more cache than this, then increase this number appropriately. If you may have celerons or other archs with crippled last-level caches, then I recommend you set CacheEdge to `4194304` (4MB). At this level, CacheEdge doesn't effectively block for caches, but it will tend to keep your workspace requirements down.

3.7.5 Handling paralellism in generic libraries

The most portable library is the serial library. You can instruct ATLAS to build only the serial library by adding `-t 0` to your configure flags.

Assuming all your platforms provide the same type of threading (eg., pthreads), then you may also build the threaded libraries. However, since ATLAS uses processor affinity, you will need to build the threaded libs to match the smallest number of processors of any machine you are targeting. You can limit the number of processors to compile for using the `-t X` configure flag, where `X` is the number of processors to tune for. So, if your smallest targeted core count was 2, you would add `-t 2` to your configure flags.

3.8 Various other flags

3.8.1 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.

3.8.2 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
```

3.8.3 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`.
- `--libdir=<dirname>` : Installation directory for ATLAS's libraries. Default: `/usr/local/atlas/lib`.

3.8.4 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 `gcc4.7.0` 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 [24] 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 2 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).

```
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 2: 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 3: Timings results when architectural defaults are compiled with substandard gcc4.1

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 3 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 4.7.0`, 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 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 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.tar.bz2` to compare against our own Core2Duo64SSE results.
3. `bunzip2 -c Core2Duo32SSE3.tar.bz2 | tar xvf -` untars the selected architectural results (replace `Core2Duo32SSE3.tar.bz2` 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 4 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.

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).

See Appendix A for details on more extensive auto-benchmarking.

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

```
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 4: Comparing 32 and 64 bit libraries on a 2.4 Ghz Core2Duo

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 --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

9.1.1 Setting up Cygwin

ATLAS 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`. Setup is usually as easy as running a install script selecting a mirror site, and selecting the right packages. If you are running a 64-bit Windows, make sure to use the 64-bit Cygwin installer! The 64-bit ATLAS install depends on Cygwin64.

If you have found you have missed a package, just rerun the install script to add it. The packages that I install are pretty much everything that mentions gcc and gfortran. You can find these packages by entering "gcc" in the search box. You will also need gfortran, and all the usual developer stuff (make, etc.). If you want to build libraries to be used by applications using MSVC++ or the Intel compilers, you will also want to be sure to install all the MinGW compilers and tools (see the following sections for more info).

If you want to compile using MinGW, then obviously you will need to install the MinGW compilers. ATLAS has been tested with the MinGW compilers provided by the Cygwin project, but should work with any legal MinGW as long as ATLAS can find them (see following sections for more info).

9.1.2 Choosing cygwin or MinGW compilers

ATLAS may be installed using either the cygwin or MinGW compilers. The cygwin compilers are usually the method of choice if you are using ATLAS exclusively within the cygwin environment. In this case, all you need to do is configure and build as normal, and the

resulting ATLAS libraries will depend on the cygwin libraries and environment in order to run. For 64-bit windows, please note that the cygwin-built ATLAS threaded libraries will depend on cygwin's pthread library, which is GPL.

However, if the user's application uses native Windows compilers such as MSVC++ or the Intel compilers, or if the resulting executable is to invoked direclty from Windows (not in a cygwin window), then you need to install ATLAS using the MinGW GNU compilers, which provide gcc/gfortran that interoperate with native compilers. To tell ATLAS to build using the MinGW compilers, you should add '`-Si nocygwin 1`' to your normal configure flags.

If you are using the MinGW compilers, configure will try to autofind the correct MinGW binutils, but this may fail. If configure does not work, you may have to specify where your MinGW binutils are installed, as discussed in Section 9.1.3.

9.1.3 Specifying the MinGW binutils to use

If ATLAS fails to find the correct MinGW compilers that you wish to use, you can manually specify complete paths to configure. In the directory where you plan to do configure, but before doing configure, create the file `MinGW64.dat` for a 64-bit install, or `MinGW32.dat` for a 32-bit install. This file specifies complete paths to all the MinGW binutils needed by ATLAS, one per line. The first line points to your MinGW `ar`, the second to `ranlib`, the third to your `gcc`, and the fourth to your `gfortran`. If you are planning to configure to not use a FORTRAN compiler (using the `--nof77` flag), then you can omit the gfortran line.

Each line must be appropriately escaped if it contains embedded spaces or parens, as Windows often does. This may or may not work, and if you have problems getting it to work, I recommend using logical links to create working paths that don't embed these characters.

Here is an example of such a file as created for a 64-bit install using the MinGW compilers provided by cygwin, saved to `BLDdir/MinGW64.dat`:

```
/usr/bin/x86_64-w64-mingw32-ar.exe
/usr/bin/x86_64-w64-mingw32-ranlib.exe
/usr/bin/x86_64-w64-mingw32-gcc.exe
/usr/bin/x86_64-w64-mingw32-gfortran.exe
```

Note that the entire file should be 4 (or 3 if gfortran isn't used) lines, with no blank lines.

9.1.4 Creating MSVC++ compatible import libraries

If you configured with `--shared`, then ATLAS should autocreate both a `.dll` and a `.def` file. My understanding is that the Windows tool `LIB` can then be used to create a MSVC++ compatible import library with commands like:

```
LIB /nologo /MACHINE:[x86,X64] /def/lib[s,t]atlas.def
```

which will create the required `.lib`. For instance:

```
LIB /nologo /MACHINE:X64 /def/libtatlas.def
```

Should create the threaded ATLAS library `libtatlas.lib` for 64-bit Windows.

You can see the genesis of this approach in the e-mail thread:

`https://sf.net/projects/math-atlas/forums/forum/1026734/topic/5349864`

9.1.5 Building 32-bit libraries on 64-bit Windows and cygwin64

I have never gotten the 32-bit gcc cygwin compilers to work under cygwin64. Therefore, if you need 32-bit libraries compiled with the cygwin compilers under Windows, I recommend installing cygwin32.

With some hand intervention, it is possible to build 32-bit ATLAS libraries under cygwin64 using the 32-bit MinGW compilers, however. To do this, do your configure as normal, but before starting the build, hand-edit the created `Make.inc`. Look for the make definition for XCCFLAGS, and you should see that it includes a `-m32`. Change this to `-m64`. You should now be able to invoke make as normal.

9.2 Special Instructions for ARM

9.2.1 Enabling NEON on 32-bit ARM

By default, ATLAS will not use ARM32's SIMD vectorization (NEON), since it is not IEEE-compliant. However, you can force its use by adding the following flags to your configure command:

```
-Fa al -mfpu=neon -Si ieee 0
```

Note that NEON only supports single precision, so this will not change accuracy or speed for double precision.

9.2.2 Special instructions for ARM big/little systems

The stable version of ATLAS is not well-suited to use big/little systems efficiently. The problem is that it can use only one block size and kernel, and one optimized for the little systems will not be optimized for the big, and vice versa. If you are only using the serial interface, the best idea is probably to install two versions, one optimized for the big and one optimized for the little CPUs.

In order to install ATLAS to run only a subsection of cores, you can use the `--force-tids` flag (see §3.3 for details). Unless you use something like `tasksel` to invoke `configure`, configure may detect the big core as the architecture, when you want the little, and vice versa. Therefore, the easist way to build a library to use only the little or only the big cores is to explicitly tell configure which architecture you want to support by using the `-A` flag to configure, as described in §3.6. ATLAS 3.10.3 currently has configure support for the following `-A` ARM32 strings: `ARMa7`, `ARMa9`, `ARMa15`, and the following ARM64 strings: `ARM64a53`, `ARM64a57`, `ARM64xgene1`. For instance, to use only the little cores on my 8-core odroid, I would enter:

```
../configure -b 32 -A ARMa7 --force-tids="4 0 1 2 3"
```

Whereas to use the big cores exclusively, I would enter:

```
../configure -b 32 -A ARMa9 --force-tids="4 4 5 6 7"
```

In order to find out which were the big and little cores, I had to examine `/proc/cpuinfo`.

The more common case is that you want to use all the parallel cores at once to do parallel BLAS calls. Given 3.10's limitations, I would recommend tuning for the small cores. The

reason is that 3.10 uses a lot of statically scheduled parallel algorithms, which means the BLAS will run at the speed of the slowest processor. Therefore, it makes sense to get the slow cores to run at peak speed, while the big cores take a performance hit by using the kernel and block factor optimized for the little cores.

So, as an example, my odroid system has 4 a7 (little) cores and 4 a9 (bit) cores. Assuming I want to use all 8 cores, but use the a7 architectural defaults, the configure line is simply:

```
../configure -b 32 -A ARMa7
```

9.2.3 ARM with inactive CPUs

Sometimes, ARM systems default to having several of the CPUs turned off to save power. If this happens, you need to turn them back on before doing the ATLAS install, as seen here:

```
http://elinux.org/Jetson/Performance
```

In case this page goes away, for my 4-core system I had to (as root):

```
echo 0 > /sys/module/cpu_tegra/parameters/auto_hotplug
echo 0 > /sys/devices/system/cpu/cpuquiet/tegra_cpuquiet/enable
echo 1 > /sys/devices/system/cpu/cpu0/online
echo 1 > /sys/devices/system/cpu/cpu1/online
echo 1 > /sys/devices/system/cpu/cpu2/online
echo 1 > /sys/devices/system/cpu/cpu3/online
```

On another system I had, the echo would not work, but copying `cpu0`'s `online` file over all others did.

9.3 Special instructions for OS X

Newer versions of OS X ship broken versions of the gnu binutils. They are deprecated in favor of **clang**, which as of 3.10.3 can be used to get a fairly performant and correct ATLAS install. Add the flag "`--force-clang=/path/to/clang`" to your configure line so that ATLAS knows clang is your native compiler. If you have a fortran compiler using the name `gfortran`, then this is all that is necessary. If it is installed in a different name (even a close one like `gfortran-5`), then you should also add: "`-C if /path/to/f77comp`". If you have no fortran compiler, see Section 3.2.4.

Note that more recent GNU **tar** releases have become incompatible with OS X's native tar. If you have difficulty untarring the tarfiles, you may need to use gnu tar rather than OS X tar. On many OS X systems, GNU tar is available as `gtar`.

9.4 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.

9.5 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] Anthony M. Castaldo and R. Clint Whaley. Minimizing Startup Costs for Performance-Critical Threading. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, pages 1–8, Rome, Italy, May 2009.

[6] Anthony M. Castaldo and R. Clint Whaley. Scaling LAPACK Panel Operations Using Parallel Cache Assignment. In *Proceedings of the 2010 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 223–231, Bangalore, India, January 2010. ACM.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

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

[14] 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.

[15] S. Toledo. Locality of reference in lu decomposition with partial pivoting. 18(4):1065–1081, 1997.

[16] R. Clint Whaley. Empirically tuning lapack's blocking factor for increased performance. In *Proceedings of the International Multiconference on Computer Science and Information Technology*, volume 3, pages 303–310, Wisla, Poland, October 2008. IEEE press.

[17] 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`.

[18] 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`.

[19] 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.

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

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

[22] 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`.

[23] 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.

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

A Post-install Timing and Benchmarking

This appendix describes how to use ATLAS's benchmarking tools, which are built and run in `OBJdir/results`.

A.1 Setting up ploticus

You will need to get ploticus installed. On ubuntu, all you need is:

```
sudo apt-get install ploticus
```

I believe the command for fedora is (as root):

```
yum install -y pl
```

For other OSes, download the software from the ploticus homepage:

```
http://ploticus.sourceforge.net/doc/download.html
```

Under ubuntu, the executable is called 'ploticus'. If your install is called something else (eg., under Fedora it is called 'pl'), then you'll need to edit your `BLDdir/results/Makefile` and change the `PLOT` macro to the correct name and path. You may also need to set where to find the ploticus prefab files. For instance, under Fedora I had to:

```
export PLOTICUS_PREFABS=/usr/share/ploticus
```

A.2 Building the existing charts

To build complete reports comparing the present ATLAS install to other LAPACK/BLAS, you can issue:

```
make atlvsys.pdf cmp="SYSTEM LIB NAME"
make atlvat2.pdf cmp="ATLASvXXXX" AT2dir="OBJdir/lib dir of previous install"
make atlvf77.pdf cmp="f77"
```

For example, to compare against MKL (setup as described below), you would issue:

```
make atlvsys.pdf cmp="MKL"
```

To compare against a previous version of ATLAS, you additionally must specify the directory of the installed libraries using the `AT2dir` macro.

You can build individual charts comparing ATLAS versus another LAPACK/BLAS install by issuing commands of the following form from `BLDdir/results`:

```
make charts/<pre><rout><side><uplo><ta>_<sz>_<cmp>[_pt].eps
```

Where the choices are:

1. `pre`: precision/type prefix, choose 's', 'd', 'c', or 'z'.
2. `rout`: choose 'mmsq' for square GEMM, 'mmrk' for rank-K update GEMM with K equal to the make macro `RK`, or one of the l3blas names: 'symm', 'herk', 'syrk', 'herk', 'syr2k', 'her2k', 'trmm', 'trsm'. You can choose one of the lapack names: 'getrf', 'potrf', 'geqrf'.

3. `side` : 'L' (Left) or 'R' (Right)
4. `uplo` : 'L' (Lower) or 'U' (Upper)
5. `ta` : 'N' (NoTranspose), 'T' (Transpose), 'C' (ConjTrans)
6. `cmp`: choose 'avs' (compare present install to system lib like ACML or MKL), 'ava' (compare against prior ATLAS install), 'avf' (compare against F77 LAPACK & BLAS).
7. `size`: varies problem sizes charted, choose: 'tin' (10-100), 'med' (200-2000), 'lrg' (2400-4000), 'cmb' (all sizes in one chart) 'mlr' (medium and large sizes in one chart).
8. `_pt`: if omitted, time serial, else time threaded

You can also get summary information that displays square GEMM and all factorization performance on one chart with the commands:

```
make charts/<pre>factor_<sz>_<lib>[_pt].eps       # results in MFLOPS
make charts/<pre>pcmm_factor_<sz>_<lib>[_pt].eps  # results as % of GEMM
```

You can also get a summary chart of all QR variants using:

```
make charts/<pre>qrvar_<sz>_<lib>[_pt].eps       # results in MFLOPS
```

Where `<lib>` is one of 'atl', 'sys', 'at2', and 'f77'.

To compare differing ATLAS installs, edit your `BLDdir/results/Make.plinc`, and set `AT2dir` to point to the other ATLAS install's `lib/` directory.

To compare against a system LAPACK/BLAS, fill in the following macros in `BLDdir/Make.inc` to point to the system libraries, rather than the default F77BLAS:

```
SBLASlib = $(FBLASlib)  # should be serial sysblas
BLASlib = $(FBLASlib)   # should be parallel sysblas
SLAPACKlib =   # set to parallel system lapack
SSLAPACKlib =  # set to serial system lapack
```

For instance, here's how to set them to use ACML:

```
SBLASlib = /opt/acml4.4.0/gfortran64/lib/libacml.a
BLASlib = /opt/acml4.4.0/gfortran64_mp/lib/libacml_mp.a -fopenmp
SLAPACKlib = /opt/acml4.4.0/gfortran64_mp/lib/libacml_mp.a
SSLAPACKlib =  $(SBLASlib)
```

MKL is a good deal more complicated, and you'll have to see Intel's directions for things to work for your setup. On mine, I did:

```
MKLROOT = /opt/intel/mkl/
SBLASlib = -Wl,--start-group $(MKLROOT)/lib/intel64/libmkl_gf_lp64.a \
           $(MKLROOT)/lib/intel64/libmkl_sequential.a \
           $(MKLROOT)/lib/intel64/libmkl_core.a -Wl,--end-group -lpthread
BLASlib = -Wl,--start-group $(MKLROOT)/lib/intel64/libmkl_gf_lp64.a \
           $(MKLROOT)/lib/intel64/libmkl_gnu_thread.a \
           $(MKLROOT)/lib/intel64/libmkl_core.a -Wl,--end-group \
           -fopenmp -lpthread
SLAPACKlib = -Wl,--start-group $(MKLROOT)/lib/intel64/libmkl_gf_lp64.a \
           $(MKLROOT)/lib/intel64/libmkl_gnu_thread.a \
           $(MKLROOT)/lib/intel64/libmkl_core.a -Wl,--end-group
SSLAPACKlib =  -Wl,--start-group $(MKLROOT)/lib/intel64/libmkl_gf_lp64.a \
           $(MKLROOT)/lib/intel64/libmkl_sequential.a \
           $(MKLROOT)/lib/intel64/libmkl_core.a -Wl,--end-group
```

After the above setup, I can compare ATLAS's medium-sized threaded Cholesky performance to that of ACML by issuing:

```
make charts/dpotrf_LLN_mlr_avs_pt.eps cmp=ACML
gv   charts/dpotrf_LLN_mlr_avs_pt.eps &
```

A.3 A guide to the tools (to build your own)

I have written a set of generic tools for manipulating the output of ATLAS's timers, and you can use and extend these tools if you want to autotime fancier/different things. All tools give usage information if you pass `--help` on the commandline. All tools default to taking input from `stdin` and output to `stdout`, so you can pipe them into each other. Each tool does a very simple thing, and the idea is you build a pipe of them to do useful work.

To make building your own tools easy, examine `SRCdir/include/atlas_tvec.h` which contains a host of prewritten routines and data structures to make tool building easy.

All the tools I have written allow you to choose to keep only certain vectors of data (corresponding to columns of output in the timer output). To give an example, say we ran the following line:

```
c2d>./xdmmtst_atl -F 120 -N 10 100 10 -T 0 -# 3 > timer.out
```

This will use `gemmtst.c` to time all square problems between 10 and 100 in steps of 10, without doing any testing, forcing at least 120MFLOPS of computation for timing accuracy, with three repetitions.

Here's the tools I have written so far:

`xatl2tvec` : Reads in the output of a timer file, and produces a standard timing vectors file that can be read by routines provided in `atlas_tvec.h` and all downstream tools. Example usage:

```
c2d>./xatl2tvec -# 3
```

`xreducetvec` : take tvec file with repetition timings and reduce them to single timings while adding simple statistics like min, max, and average.

`xcattvecs` : take multiple vector files and combine them into one file for later comparison. Renames vectors as necessary by adding `_#` to repeated names coming from later files. Can specify for some vectors to get this statistical treatment, and other vectors to just use the first one found.

`xtvec2plp` : Take a standard tvec file and produce a standard `ploticus` data file from it.

`xmergetvecs` : Take two standard tvecs that contain separate runs of the same data with non-overlapping data, and combine them into one vector. Eg., you do one run with $N = 100, 200, 300$ and a second with $N = 1000, 5000, 8000$. This routine will allow you to combine these N ranges into one for charting all results in one graph. This can be done repeatedly to merge any number of runs together.

`xperctvecs` : recast named tvecs as a percentage of a baseline. Can also be used to compute speedup rather than percentage by adding `-m 1.0` flag.

To see how these tools can be used together, you can trace the dependence chain of any of the charts that are autobuilt, as explained in §A.2.