Moving SCA/LAPACK to Higher Precision

without *too much* work

Yozo Hida

yozo@cs.berkeley.edu

Computer Science Division
EECS Department
U.C. Berkeley

April 28, 2008

Outline

Outline

Motivation for higher precision

- LAPACK and SCALAPACK are widely used.
- User survey revealed small but significant portion of users wanted precision higher than double precision.
  `http://icl.cs.utk.edu/lapack-forum/survey/`
- We want to support codes like

  n_digits $\leftarrow 32$
  repeat
    n_digits $\leftarrow$ n_digits $\times 2$
    Solve with n_digits
  until error $<$ tol
- We want to do the least amount of work to support multiple precision.

Fast CPU, slow memory

Gap between CPU performance and memory speed has been increasing exponentially.

What can we do with more computational power?

Fast CPU, slow memory

Gap between CPU performance and memory speed has been increasing exponentially.

What can we do with more computational power?

Solve bigger problems.

- Memory bandwidth / latency is often the bottleneck.

Fast CPU, slow memory

Gap between CPU performance and memory speed has been increasing exponentially.

What can we do with more computational power?

Solve bigger problems.

- Memory bandwidth / latency is often the bottleneck.

Do more computation per unit data.

- solve problems more accurately,
- debug problems without waiting too long.
  - Numerical instability can be ameliorated with higher precision.

More work per unit data

- Speeds of matrix-vector multiply:

	time (s)	mop/s	time ratio
double (MKL)	0.052	965	1
double-double	0.27	189	5.1
quad-double	5.85	8.55	113

Dimension 5000.
Ran on 1 core of dual quad-core "Cloverton" Intel Xeon 2.33 GHz.

  - Double-double uses 17.5 times more flops but only twice as much data. Matrix-vector multiply runs only 5.1 times slower.
  - Quad-double uses approx. 200 times more flops but only four times as much data. Matrix-vector multiply only runs 113 times slower.

- Greater effect for sparse and parallel codes, where flops is even "cheaper".

Outline

Some Implementations of Higher Precision

- Fixed precision
  - Compiler supported quad
  - Double-double, Quad-double
- Arbitrary precision
  - GMP / MPFR
  - ARPREC

Double-double and quad-double numbers

Store higher precision numbers as an unevaluated sum.
`http://www.cs.berkeley.edu/~yozo/`

- $2^{100}+1$ is stored as the pair $(2^{100}, 1)$ in double-double.
- Can represent about 32 / 64 digits of precision,
- Highly efficient algorithms due to their fixed, small size.
- Simple representation: fixed size array of doubles.
  Makes it easy to allocate arrays and use MPI.
- $\pi \approx 3.14159265358979323846264338327950288419716939937510582097494$
  is stored as

$$\begin{array}{rll}
3.1415926535897931160 \times 10^{0} & (\quad 1 & 1100100100001111110110101010001000100001011010001100\\
1.2246467991473532072 \times 10^{-16} & (-\ 53 & 1000110100110001001100011001100010100010111000000011\\
-2.9947698097183396659 \times 10^{-33} & (-109 & 1111100011001011101101011011111101101100011111011110\\
1.1124542208633652815 \times 10^{-49} & (-163 & 1010011001111100110001110100000000100000101110111110\\
\end{array}$$

  in quad-double.
- Peculiar precision: $1+2^{-1000} \neq 1$ in double-double arithmetic.

Double-double and quad-double numbers

- C, C++ and Fortran 95 interfaces.

```
use qdmodule
type (qd_real) a, b
a = 1.d0
b = cos(a)**2 + sin(a)**2 - 1.d0
call qdwrite(6, b)
```

- Support for complex data types recently added.

GMP/MPFR

- Multiple-precision floating-point computations with correct rounding.
- `http://www.mpfr.org/`
- Uses integer arithmetic instructions.
- Highly optimized for variety of platforms.
- Somewhat complicated data structure, mixing various types in a C struct.
  This makes inter-language operation, porting, and message passing somewhat harder.
- C, C++ interfaces. No Fortran 95 interface.

ARPREC

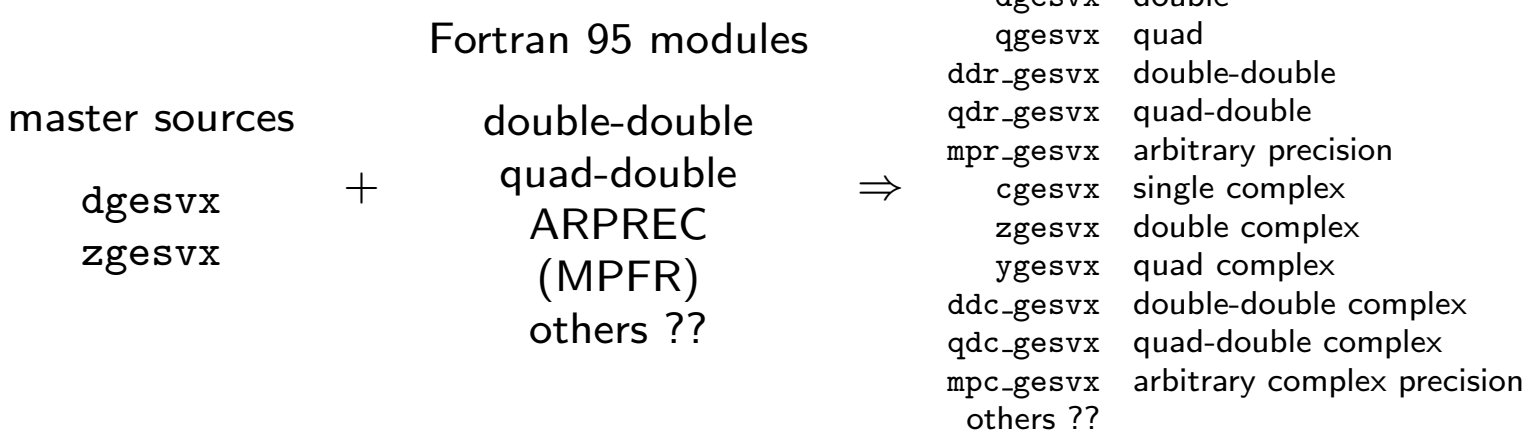
- `http://www.cs.berkeley.edu/~yozo/`
- Uses simple floating point array to represent data.
  This makes inter-language operation and message passing easier.
- Slower than GMP.
- C, C++, and Fortran 95 interfaces.

```
use mpmodule
type (mp_real) a, b
a = 1.d0
b = cos(a)**2 + sin(a)**2 - 1.d0
call mpwrite(6, b)
```

Outline

Source-to-source translator

- LAPACK contains $^3/_4$ million lines of FORTRAN 77.
  We do not want to rewrite LAPACK.
- Source-to-source translation:



Things to Consider

- Single source code for varying precision. This makes it much easier to maintain code.
- Workspace allocation. We need to tell the user how much workspace to allocate.
- Temporary variable allocation.
- Should we allow multiple precisions in one routine? Within one matrix?
- How to adjust precision.
- Can multiple versions co-exist? Naming issues.

Modules

The high precision library provides a module that declares what operators and functions are overloaded:

```
module mpmodule
  type mp_real
    sequence
    real*8 mpr(mpwds+5)
  end type

  interface operator (+)
    module procedure mpadd
  end interface

...

end module mpmodule
```

Modules

LAPACK source is then modified slightly:

```
subroutine some_lapack_routine
  !  Use module for arbitrary precision
  use mpmodule

  !  Declare variables in the desired format
  type (mp_real) a, b, c

  !  LAMCH must be provided by the
  !  arbitrary precision package provider
  smlnum = lamch(a, 'Safe minimum')

  !  ...the rest of the code ...
end subroutine
```

Assessment of Constants

Simple replacement of variable type is not enough:

```
x = 0.1d0 / 0.3d0
```

This line is interpreted by the compiler as double precision constants and computation. Needs to be replaced with something like,

```
x = mp_real("0.1") / mp_real("0.3")
```

This is the most common "bugs" reported to us when people convert their code to use our higher precision packages.

Naming Issues

How should a routine be named when using a new precision?

- Add a new prefix. DGESVX becomes QGESVX for quad, QD_GESVX for quad-double etc.
  Leads to name explosion, but easier for inter-language operations.
- Use generic names like "GESVX" and dispatch to correct routines using Fortran 95 module and interface facilities. LAPACK 95 does this.

Implementation

Currently implemented as a (rather ad-hoc) Perl script to convert LAPACK sources to perform

- use appropriate module file (provided by the dd / qd / arprec packages),
- constant literal substitutions,
- handle Fortran data declarations,
- use a new prefix for each LAPACK routines,
- substitute something appropriate for read / write statements, and
- declare variables in appropriate types.

Passes most LAPACK tests for Quad and QD precisions, including linear systems.

Current Work

More complete Fortran parser.

- Open Fortran Parser. Supports Fortran 2003.
- Parser from `ftnchek`. Mostly for Fortran 77.
- Gfortran or G95 frontend.

Variable precision LAPACK using ARPREC.

Outline

Conclusion

Once the high-precision library provides the appropriate module file declaring the overridden functions, most of LAPACK code can be transformed to use them without too much effort.

Future Work

- Using ARPREC in LAPACK (current).
- Use more robust Fortran parser (current).
- Facilities for debugging numerical problems.
- Exception handling.
- How much performance can we buy from using multi-core BLAS?
- Apply same technique to SCALAPACK.
- Provide F95 interface to MPFR. Anyone?