Linear Algebra Studies

Erik Myklebust: comparative study of CLHEP, BLAS/LAPACK, GSL, uBLAS (BOOST)
http://seal.web.cern.ch/seal/snapshot/work-packages/mathlibs/index.html
Context: Kalman filtering for track reconstruction

presentation by Matthias Winkler

About this presentation

- problem domain: Kalman filter equations for update of track state

- memory managment: heap vs. stack allocation
  understand individual memory managment
  of CLHEP, GSL and uBLAS

- expression templates: avoid temporaries

- timing: Pentium rdtsc()

- results and comments

Erik built a wrapper based on expression templates for
BLAS/LAPACK and GSL

Equations (Expressions)

Update of the track state vector:

$$x = x' + K(m - Hx')$$

Calculation of the Kalman gain matrix:

$$K = C'H^T(V + HCH^T)^{-1}$$

Update of the track state errors (covariance matrix):

$$C = (I - KH)C'$$

Calculation of the local Chi2 increment:

$$Chi2 = (m - Hx')^T(V + HC'H^T)^{-1}(m - Hx')$$

where x, x' track state vector, 5-dimensional
- x updated state (new estimate)
- x' predicted state

m measurement vector, 2-dimensional
K Kalman gain matrix, 5x2
H projection matrix H = dm/dx, 2x5
V measurement errors, 2x2 symmetric matrix
C, C' covariance matrices, 5x5 symmetric matrices
- C updated state (new estimate)
- C' predicted state

Memory managment

CLHEP: - allocate fixed size array of 25 elements
at instantiation
- if(n < 25): use array (or part of it)
- else: allocate from the heap
(and don't use 25 elements array at all)

GSL: gsl_allocate (malloc in the end)

uBLAS: stack + heap

BLAS/LAPACK wrapper: heap (new/delete)

in addition: packed vs. unpacked storage format of
symmetric matrices

note: allocation of one double on the HEAP costs ~2000 cpu clock cycles on a Pentium4!

Expression templates

CLHEP (classic operator overloading):

```
Matrix operator*(const Matrix& a, const Matrix& b) {

    Matrix c;
    for(i = 0; ...)
        for(j = 0; ...)
            c(i,j) = ...

    return c;
}
```

Expression templates:

```
MatrixExpression operator*(const Matrix& a, const Matrix& b) {

    return MatrixExpression(MMProd(a,b));
}
```

Expression Templates Technicalities

- operator overloading and templated operators

- conversion constructors and implicit conversion

- templated constructors and templated methods

- template specialization

About the timer

- select a timer with minimal overhead

- found PentiumTimer from COBRA (Vincenzo Innocente)

- reads Pentium *time stamp counter* register with rdtsc() assembly function

- overhad for two consecutive reads: 80-100 clock cycles

Hardware and software technical specifications

Hardware:

Intel Pentium4 (8 kB L1 cache, 512 kB L2 cache)
1.8 and 2.4 GHz
256MB RAM

Software:

Standard CERN Linux installation (CERN Linux Redhat 7.3)
Software configuration: SEAL_0_3_1
Compilers: gcc version 3.22, g77
Compiler flags:
- standard optimization (-O2) and shared libraries (lib*.so)
- special optimization
  (-mcpu=pentium4 -march=pentium4 -msse2 -O3)
  and archive libraries (lib*.a) for BLAS/LAPACK.

Results

1...update of state vector x
2...calculation of Kalman gain matrix K
3...update of state error C
4...calculation of local Chi2 increment (not part of Kalman filter step)

numbers are in “clock_ticks (relative_to_CLHEP)”

	CLHEP	BLAS/L.	GSL	uBLAS, heap	uBLAS, stack
1	2,334	3,117 (1.34)	6,716 (2.88)	3,342 (1.43)	1,827 (0.78)
2	9,330	14,800 (1.59)	25,950 (2.78)	136,900 (14.67)	144,100 (15.44)
3	6,223	6,092 (0.98)	8,395 (1.35)	15,920 (2.56)	15,090 (2.42)
4	5,893	12,500 (2.12)	20,000 (3.39)	skipped	skipped
1+2+3	18,760	25,090 (1.34)	46,670 (2.49)	163,300 (8.70)	189,000 (9.20)

General comments

- significant differences between libraries

- higher gcc optimization flags do not necessarily result in faster executables

- uBlas performance depends on the way expressions are written, in one line (relatively slow) or over many lines (relatively fast)

- GSL 1.3-2x slower than BLAS/LAPACK

Why the BLAS/LAPACK wrapper is slower than CLHEP

1) The BLAS/LAPACK wrapper uses dynamic memory allocation on the heap. For the steps 1, 2 and 3 one needs to allocate 8 times memory: 2 allocations in step 1,
   4 allocations in step 2 (if the transpose of H is available, otherwise one temporary more needs to be crated)
   2 allocations again in step 3.
   Only these memory allocations consume about 14500 clockticks which is more than half of the number of clockticks of the result in 5!
   These memory allocations are negligible for CLHEP, as all matrices and vectors need less than 25 elements and therefore the pre-allocated memory from the stack is used.

2) The inversion of 2x2 symmetric matrices is much faster for CLHEP than it is for LAPACK (164 versus 2100), as hand-optimized inversion algorithms are implemented in CLHEP for matrices up to dimension 6x6. The numerical stability of these algorithms could not be tested within the available time.

Obvious improvements for BLAS/LAPACK wrapper

- use stack allocation if size is known in advance
  or
  use dedicated stack allocator for dynamic memory allocation

- optimized 2x2 sym. matrix inversion algorithm instead of LAPACK one

Even if CLHEP was fastest in this test the BLAS/LAPACK wrapper can be improved such that it leads to a better performance than CLHEP. Changes should be primarily done in the way of how memory allocation is done.

General comments

compared CLHEP, BLAS/LAPACK, GSL and uBLAS by measuring the time it takes of typical Kalman filter update step of a particle track state (vectors of size 2 and 5, symmetric matrices 2x2 and 5x5, projection matrix 2x5, Kalman gain matrix 5x2)

1) It is possible to write a C++ wrapper for libraries written in C or Fortran with a small overhead (3%-8%).
2) BLAS/LAPACK wrapper:
   - main amount of time is lost in allocating and deallocating memory
   - time necessary for carrying out calculations is less than half of the total time
3) CLHEP was the fastest library because of its internal memory management
   it is expected that the performance of CLHEP drops significantly if working with expressions of algebraic objects of higher dimensions (>25 elements).
4) GSL always slower than CLHEP and BLAS/LAPACK
   - both GSL implementation of BLAS and the GSL memory management are the main reasons
5) uBLAS slowest of all, up to 10x
   - unknown reasons

For the future

1) test other compilers: gcc-3.4, icc/ecc etc.

2) use optimized BLAS for Pentium/Linux (such as IMK)

3) use optimized BLAS for small matrices/vectors

3) combining algebraic objects of different precision types
(only supported by uBLAS)

4) hand-optimized (inversion-) algorithms for small size problems

5) test other platforms/OS (64 bit processors)