Solving Triangular Systems

1. Conditioning and Multiple Double Arithmetic
   - ill conditioned matrices
   - quad double arithmetic
2. On a Parallel Shared Memory Computer
   - rewriting the formulas
   - a parallel solver with OpenMP
3. Accelerated Back Substitution
   - partitioning an upper triangular system in tiles
   - experimental results

MCS 572 Lecture 27
Introduction to Supercomputing
Jan Verschelde, 28 October 2024

Solving Triangular Systems

1. Conditioning and Multiple Double Arithmetic
   - ill conditioned matrices
   - quad double arithmetic
2. On a Parallel Shared Memory Computer
   - rewriting the formulas
   - a parallel solver with OpenMP
3. Accelerated Back Substitution
   - partitioning an upper triangular system in tiles
   - experimental results

ill conditioned matrices

Consider the 4-by-4 lower triangular matrix

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -2 & 1 & 0 & 0 \\ -2 & -2 & 1 & 0 \\ -2 & -2 & -2 & 1 \end{bmatrix}.$$

What we know from numerical analysis:

1. The condition number of a matrix magnifies roundoff errors.
2. The hardware double precision is $2^{-52} \approx 2.2 \times 10^{-16}$.
3. We get no accuracy from condition numbers larger than 10^{16}.

an experiment in an interactive Julia session

```
julia> using LinearAlgebra

julia> A = ones(32,32);

julia> D = Diagonal(A);

julia> L = LowerTriangular(A);

julia> LmD = L - D;

julia> L2 = D - 2*LmD;

julia> cond(L2)
2.41631630569077e16
```

The condition number is estimated at 2.4×10^{16}.

Solving Triangular Systems

1. Conditioning and Multiple Double Arithmetic
   - ill conditioned matrices
   - quad double arithmetic

2. On a Parallel Shared Memory Computer
   - rewriting the formulas
   - a parallel solver with OpenMP

3. Accelerated Back Substitution
   - partitioning an upper triangular system in tiles
   - experimental results

quad double arithmetic

A quad double is an unevaluated sum of 4 doubles,
improves working precision from 2.2×10^{-16} to 2.4×10^{-63}.

Y. Hida, X.S. Li, and D.H. Bailey: **Algorithms for quad-double precision floating point arithmetic.** In *15th IEEE Symposium on Computer Arithmetic* pages 155–162. IEEE, 2001. Software at `http://crd.lbl.gov/~dhbailey/mpdist/qd-2.3.17.tar.gz`.

A quad double builds on `double double`, some features:

- The least significant part of a `double double` can be interpreted as a compensation for the roundoff error.
- Predictable overhead: working with `double double` is of the same cost as working with complex numbers.

operator overloading in C++

```
#include <iostream>
#include <iomanip>
#include <qd/qd_real.h>
using namespace std;

int main ( void )
{
   qd_real q("2");
   cout << setprecision(64) << q << endl;
   for(int i=0; i<8; i++)
   {
      qd_real dq = (q*q - 2.0)/(2.0*q);
      q = q - dq; cout << q << endl;
   }
   cout << scientific << setprecision(4);
   cout << "residual : " << q*q - 2.0 << endl;
   return 0;
}
```

compiling with a makefile

On `pascal`, the makefile contains the entry:

```
QD_ROOT=/usr/local/qd-2.3.17
QD_LIB=/usr/local/lib

qd4sqrt2:
        g++ -I$(QD_ROOT)/include qd4sqrt2.cpp \
            $(QD_LIB)/libqd.a \
            -o qd4sqrt2
```

Compiling at the command prompt `$`:

```
$ make qd4sqrt2
g++ -I/usr/local/qd-2.3.17/include qd4sqrt2.cpp \
            /usr/local/lib/libqd.a \
            -o qd4sqrt2
```

running the code

```
$ ./qd4sqrt2
2.0000000000000000000000000000000000000000000000000000000000000000e+00
1.5000000000000000000000000000000000000000000000000000000000000000e+00
1.4166666666666666666666666666666666666666666666666666666666666667e+00
1.4142156862745098039215686274509803921568627450980392156862745098e+00
1.4142135623746899106262955788901349101165596221157440445849050192e+00
1.4142135623730950488016896235025302436149819257761974284982894987e+00
1.4142135623730950488016887242096980785696718753772340015610131332e+00
1.4142135623730950488016887242096980785696718753769480731766797380e+00
1.4142135623730950488016887242096980785696718753769480731766797380e+00
residual : 0.0000e+00
$
```

General multiple double arithmetic is available:

M. Joldes, J.-M. Muller, V. Popescu, W. Tucker.
CAMPARY: Cuda Multiple Precision Arithmetic Library and Applications.
In *Mathematical Software – ICMS 2016, the 5th International Conference on Mathematical Software*, pages 232–240, Springer-Verlag, 2016.

Numerical Conditioning and Stability

The condition number of a random number is known to grow exponentionally in the dimension, *almost surely*, as demonstrated in

- D. Viswanath and L. N. Trefethen.
  Condition numbers of random triangular matrices.
  SIAM J. Matrix Anal. Appl., 19(2):564–581, 1998.

The numerical stability of various parallel algorithms to solve triangular systems is discussed in

- N. J. Higham.
  Stability of parallel triangular system solvers.
  SIAM J. Sci. Comput., 16(2):400–413, 1995.

Solving Triangular Systems

1. Conditioning and Multiple Double Arithmetic
   - ill conditioned matrices
   - quad double arithmetic
2. On a Parallel Shared Memory Computer
   - rewriting the formulas
   - a parallel solver with OpenMP
3. Accelerated Back Substitution
   - partitioning an upper triangular system in tiles
   - experimental results

formulas for forward substitution

Expanding the matrix-vector product $L\mathbf{y}$ in $L\mathbf{y} = \mathbf{b}$ leads to

$$\left\{ \begin{array}{lcl} y_1 & = & b_1 \\ \ell_{2,1} y_1 + y_2 & = & b_2 \\ \ell_{3,1} y_1 + \ell_{3,2} y_2 + y_3 & = & b_3 \\ & \vdots & \\ \ell_{n,1} y_1 + \ell_{n,2} y_2 + \ell_{n,3} y_3 + \cdots + \ell_{n,n-1} y_{n-1} + y_n & = & b_n \end{array} \right.$$

and solving for the diagonal elements gives

$$\begin{array}{rcl} y_1 & = & b_1 \\ y_2 & = & b_2 - \ell_{2,1} y_1 \\ y_3 & = & b_3 - \ell_{3,1} y_1 - \ell_{3,2} y_2 \\ & \vdots & \\ y_n & = & b_n - \ell_{n,1} y_1 - \ell_{n,2} y_2 - \cdots - \ell_{n,n-1} y_{n-1} \end{array}$$

rewriting the formulas

Solving $L\mathbf{y} = \mathbf{b}$ for $n = 5$:

1. $\mathbf{y} := \mathbf{b}$
2. $y_2 := y_2 - \ell_{2,1} \star y_1$
   $y_3 := y_3 - \ell_{3,1} \star y_1$
   $y_4 := y_4 - \ell_{4,1} \star y_1$
   $y_5 := y_5 - \ell_{5,1} \star y_1$
3. $y_3 := y_3 - \ell_{3,2} \star y_2$
   $y_4 := y_4 - \ell_{4,2} \star y_2$
   $y_5 := y_5 - \ell_{5,2} \star y_2$
4. $y_4 := y_4 - \ell_{4,3} \star y_3$
   $y_5 := y_5 - \ell_{5,3} \star y_3$
5. $y_5 := y_5 - \ell_{5,4} \star y_4$

$\mathbf{y} := \mathbf{b}$;
for i from 2 to n do
  for j from i to n do
    $y_j := y_j - \ell_{j,i-1} \star y_{i-1}$

$\Rightarrow$ all instructions in the j loop are independent from each other!

data parallelism

Consider the inner loop in the algorithm to solve $L\mathbf{y} = \mathbf{b}$:

$\mathbf{y} := \mathbf{b}$;
for i from 2 to n do

for j from i to n do
$y_j := y_j - \ell_{j,i-1} \star y_{i-1}$;

We distribute the update of $y_i, y_{i+1}, \ldots, y_n$ among p processors.

If $n \gg p$, then we expect a close to optimal speedup.

Solving Triangular Systems

1. Conditioning and Multiple Double Arithmetic
   - ill conditioned matrices
   - quad double arithmetic
2. On a Parallel Shared Memory Computer
   - rewriting the formulas
   - a parallel solver with OpenMP
3. Accelerated Back Substitution
   - partitioning an upper triangular system in tiles
   - experimental results

a parallel solver

For our parallel solver for triangular systems:

- For $L = [\ell_{i,j}]$, we generate random numbers for $\ell_{i,j} \in [0,1]$.
  The exact solution $\mathbf{y}$: $y_i = 1$, for $i = 1, 2, \ldots, n$.
  We compute the right hand side $\mathbf{b} = L\mathbf{y}$.
- Even already in small dimensions,
  the condition number may grow exponentially.

  Hardware double precision is insufficient.

  Therefore, we use quad double arithmetic.
- We use a straightforward OpenMP implementation.

solving random lower triangular systems

Relying on hardware doubles is problematic:

```
$ time ./trisol 10
last number : 1.0000000000000009e+00

real    0m0.003s    user    0m0.001s    sys    0m0.002s

$ time ./trisol 100
last number : 9.9999999999974221e-01

real    0m0.005s    user    0m0.001s    sys    0m0.002s

$ time ./trisol 1000
last number : 2.7244600009080568e+04

real    0m0.036s    user    0m0.025s    sys    0m0.009s
```

a matrix of quad doubles

Allocating data in the main program:

```
{
   qd_real b[n],y[n];

   qd_real **L;
   L = (qd_real**) calloc(n,sizeof(qd_real*));
   for(int i=0; i<n; i++)
      L[i] = (qd_real*) calloc(n,sizeof(qd_real));

   srand(time(NULL));
   random_triangular_system(n,L,b);
```

a random triangular system

```
void random_triangular_system
 ( int n, qd_real **L, qd_real *b )
{
   for(int i=0; i<n; i++)
   {
      L[i][i] = 1.0;
      for(j=0; j<i; j++)
      {
         double r = ((double) rand())/RAND_MAX;
         L[i][j] = qd_real(r);
      }
      for(int j=i+1; j<n; j++)
         L[i][j] = qd_real(0.0);
   }
   for(int i=0; i<n; i++)
   {
      b[i] = qd_real(0.0);
      for(int j=0; j<n; j++)
         b[i] = b[i] + L[i][j];
   }
}
```

solving the system

```
void solve_triangular_system_swapped
 ( int n, qd_real **L, qd_real *b, qd_real *y )
{
   for(int i=0; i<n; i++) y[i] = b[i];

   for(int i=1; i<n; i++)
   {
      for(int j=i; j<n; j++)
         y[j] = y[j] - L[j][i-1]*y[i-1];
   }
}
```

using OpenMP

```
void solve_triangular_system_swapped
 ( int n, qd_real **L, qd_real *b, qd_real *y )
{
   int j;

   for(int i=0; i<n; i++) y[i] = b[i];

   for(int i=1; i<n; i++)
   {
      #pragma omp parallel shared(L,y) private(j)
      {
         #pragma omp for
         for(j=i; j<n; j++)
            y[j] = y[j] - L[j][i-1]*y[i-1];
      }
   }
}
```

experimental timings

`time ./trisol_qd_omp n p` on 12-core Intel X5690, 3.47 GHz, for dimension $n = 8,000$, for varying number p of cores.

p	cpu time	real	user	sys
1	21.240s	35.095s	34.493s	0.597s
2	22.790s	25.237s	36.001s	0.620s
4	22.330s	19.433s	35.539s	0.633s
8	23.200s	16.726s	36.398s	0.611s
12	23.260s	15.781s	36.457s	0.626s

The serial part is the generation of the random numbers for L and the computation of $\mathbf{b} = L\mathbf{y}$. Recall Amdahl's Law.

We can compute the serial time, subtracting for $p = 1$, from the real time the cpu time spent in the solver, i.e.: $35.095 - 21.240 = 13.855$. For $p = 12$, time spent on the solver is $15.781 - 13.855 = 1.926$. Compare 1.926 to 21.240/12 = 1.770.

Solving Triangular Systems

1. Conditioning and Multiple Double Arithmetic
   - ill conditioned matrices
   - quad double arithmetic

2. On a Parallel Shared Memory Computer
   - rewriting the formulas
   - a parallel solver with OpenMP

3. Accelerated Back Substitution
   - partitioning an upper triangular system in tiles
   - experimental results

partitioning an upper triangular system in tiles

Consider a 3-by-3-tiled upper triangular system $U\mathbf{x} = \mathbf{b}$

$$U = \begin{bmatrix} U_1 & A_{1,2} & A_{1,3} \\ & U_2 & A_{2,3} \\ & & U_3 \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \mathbf{x}_3 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \\ \mathbf{b}_3 \end{bmatrix},$$

where U_1, U_2, U_3 are upper triangular, with nonzero diagonal elements.

Invert all diagonal tiles: $\begin{bmatrix} U_1^{-1} & A_{1,2} & A_{1,3} \\ & U_2^{-1} & A_{2,3} \\ & & U_3^{-1} \end{bmatrix}$.

- The inverse of an upper triangular matrix is upper triangular.
- Solve an upper triangular system for each column of the inverse.
- The columns of the inverse can be computed independently.

$\Rightarrow$ Solve many smaller upper triangular systems in parallel.

the second stage

Solve $U\mathbf{x} = \mathbf{b}$ for $U = \left[\begin{array}{ccc} U_1^{-1} & A_{1,2} & A_{1,3} \\ & U_2^{-1} & A_{2,3} \\ & & U_3^{-1} \end{array}\right]$

in the following steps:

$$\begin{array}{rlcl} 1) & \mathbf{x}_3 & := & U_3^{-1}\mathbf{b}_3, \\ 2) & \mathbf{b}_2 & := & \mathbf{b}_2 - A_{2,3}\mathbf{x}_3, \ \ \mathbf{b}_1 \ := \ \mathbf{b}_1 - A_{1,3}\mathbf{x}_3, \\ 4) & \mathbf{x}_2 & := & U_2^{-1}\mathbf{b}_2, \\ 5) & \mathbf{b}_1 & := & \mathbf{b}_1 - A_{1,2}\mathbf{x}_2, \\ 6) & \mathbf{x}_1 & := & U_1^{-1}\mathbf{b}_1. \end{array}$$

Statements on the same line can be executed in parallel.

In multiple double precision, several blocks of threads collaborate in the computation of one matrix-vector product.

two stages, three kernels

Algorithm 1: TILED ACCELERATED BACK SUBSTITUTION.

Input : N is the number of tiles,
n is the size of each tile,
U is an upper triangular Nn-by-Nn matrix,
$\mathbf{b}$ is a vector of size Nn.

Output : $\mathbf{x}$ is a vector of size Nn: $U\mathbf{x} = \mathbf{b}$.

1. Let U_1, U_2, ..., U_N be the diagonal tiles.
   The kth thread solves $U_i \mathbf{v}_k = \mathbf{e}_k$, computing the kth column U_i^{-1}.
2. For $i = N, N-1, \ldots, 1$ do
   1. n threads compute $\mathbf{x}_i = U^{-1}\mathbf{b}_i$;
   2. simultaneously update $\mathbf{b}_j$ with $\mathbf{b}_j - A_{j,i}\mathbf{x}_i$,
      $j \in \{1, 2, \ldots, i-1\}$ with $i-1$ blocks of n threads.

A parallel execution could run in time proportional to Nn.

data staging

A matrix U of multiple doubles is stored as $[U_1, U_2, \ldots, U_m]$,

- U_1 holds the most significant doubles of U,
- U_m holds the least significant doubles of U.

Similarly, $\mathbf{b}$ is an array of m arrays $[\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_m]$,
sorted in the order of significance.

In complex data, real and imaginary parts are stored separately.

The main advantages of this representation are twofold:

1. facilitates staggered application of multiple double arithmetic,
2. benefits efficient memory coalescing,
   as adjacent threads in one block of threads read/write adjacent data in memory, avoiding bank conflicts.

Solving Triangular Systems

1. Conditioning and Multiple Double Arithmetic
   - ill conditioned matrices
   - quad double arithmetic
2. On a Parallel Shared Memory Computer
   - rewriting the formulas
   - a parallel solver with OpenMP
3. Accelerated Back Substitution
   - partitioning an upper triangular system in tiles
   - experimental results

experimental setup

About the input matrices:

- Random numbers are generated for the input matrices.
- Condition numbers of random triangular matrices almost surely grow exponentially [Viswanath & Trefethen, 1998].
- In the standalone tests, the upper triangular matrices are the Us of an LU factorization of a random matrix, computed by the host.

Two input parameters are set for every run:

- The size of each tile is the number of threads in a block.
  The tile size is a multiple of 32.
- The number of tiles equals the number of blocks.
  As the V100 has 80 streaming multiprocessors,
  the number of tiles is at least 80.

back substitution on the V100, milliseconds, Gigaflops

double double precision

stage in Algorithm 1	64×80	128×80	256×80
invert diagonal tiles	1.2	9.3	46.3
multiply with inverses	1.7	3.3	8.9
back substitution	7.9	4.7	12.2
time spent by kernels	5.0	17.3	67.4
wall clock time	82.0	286.0	966.0
kernel time flops	190.6	318.7	525.1
wall clock flops	11.7	19.2	36.7

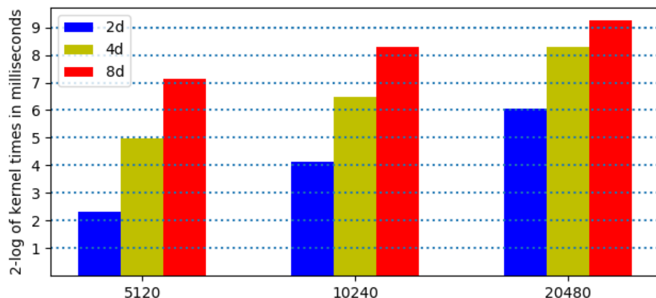
quad double precision

stage in Algorithm 1	64×80	128×80	256×80
invert diagonal tiles	6.2	38.3	137.4
multiply with inverses	12.2	23.8	63.1
back substitution	13.3	26.7	112.2
time spent by kernels	31.7	88.8	312.7
wall clock time	187.0	619.0	2268.0
kernel time flops	299.4	614.2	1122.3
wall clock flops	50.8	88.1	154.8

2-logarithms of times on the V100 in 3 precisions

Consider the doubling of the dimension and the precision.

1. Double the dimension, expect the time to quadruple.
2. From double double to quad double: 11.7 is multiplier, from quad double to octo double: 5.4 times longer.

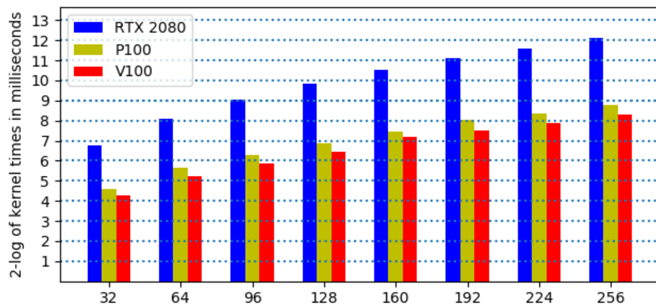


The heights of the bars are closer to each other in higher dimensions.

kernel times in quad double precision on 3 GPUs

The V100 has 80 multiprocessors,
its theoretical peak performance is 1.68 times that of the P100.

The value for N is fixed at 80, n runs from 32 to 256:



Observe the heights of the bars as the dimensions double
and the relative performance of the three different GPUs.

$20480 = 320 \times 64 = 160 \times 128 = 80 \times 256$

Back substitution in quad double precision, for $20480 = N \times n$, for three different combinations of N and n, on the V100.

stage in Algorithm 1	320×64	160×128	80×256
invert diagonal tiles	13.5	35.8	132.3
multiply with inverses	49.0	47.5	64.3
back substitution	84.6	91.7	112.3
time spent by kernels	147.1	175.0	308.9
wall clock time	2620.0	2265.0	2071.0
kernel time flops	683.0	861.1	1136.1
wall clock flops	38.3	66.5	169.5

The units of all times are milliseconds, flops unit is Gigaflops.

recommended reading

- D. H. Heller.
  A survey of parallel algorithms in numerical linear algebra.
  SIAM Review, 20(4):740–777, 1978.

- W. Nasri and Z. Mahjoub.
  Optimal parallelization of a recursive algorithm for triangular matrix inversion on MIMD computers.
  Parallel Computing, 27:1767–1782, 2001.

- J. Verschelde.
  Least squares on GPUs in multiple double precision.
  In *The 2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 828–837. IEEE, 2022.
  Code at `github.com/janverschelde/PHCpack/src/GPU`.

Exercises

1. Write a parallel solver with OpenMP to solve $U\mathbf{x} = \mathbf{y}$.
   Take for U a matrix with random numbers in $[0, 1]$, compute $\mathbf{y}$ so all components of $\mathbf{x}$ equal one. Test the speedup of your program, for large enough values of n and a varying number of cores.
2. Describe a parallel solver for upper triangular systems $U\mathbf{y} = \mathbf{b}$ for distributed memory computers. Write a prototype implementation using MPI and discuss its scalability.
3. Consider a tiled lower triangular system $L\mathbf{x} = \mathbf{b}$.
   Develop a parallel solver with OpenMP, or with Julia.