Mixed Precision Iterative Methods using High Precision Arithmetic

Hidehiko Hasegawa
hasegawa@slis.tsukuba.ac.jp

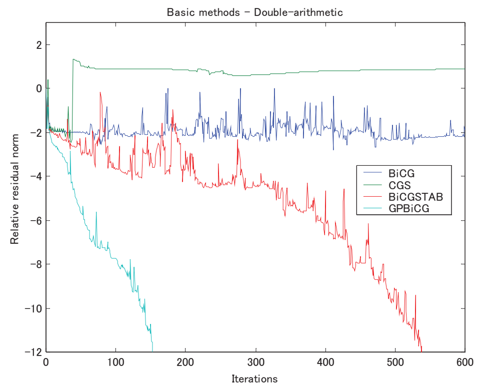
Faculty of Library, Information and Media Science,
University of Tsukuba

Accelerate Iterative Methods

- Good Algorithms
- Good Preconditioners
- Parallel Algorithms
- Good Implementations
- Accurate Computations

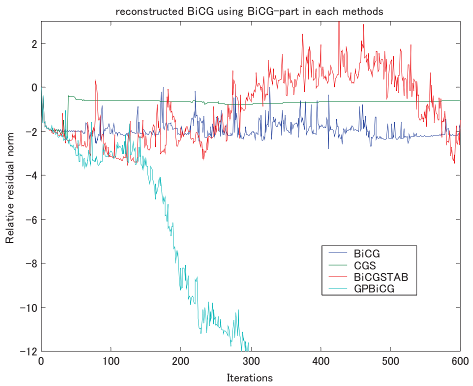
Numerical Comparison of Accelerating Polynomials in Product-type Iterative Methods

Convergence history



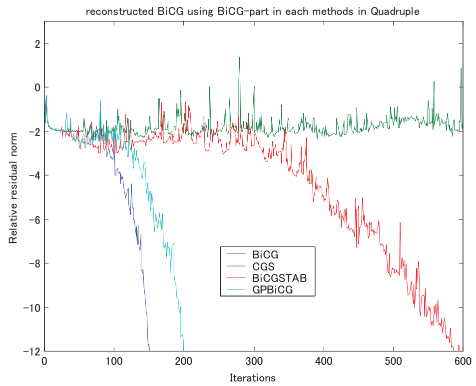
Convergence history of Bi-CG part

(reconstruct Bi-CG using alpha and beta in each methods)



Convergence of Bi-CG part: Quadruple

(reconstruct Bi-CG using alpha and beta in each methods)

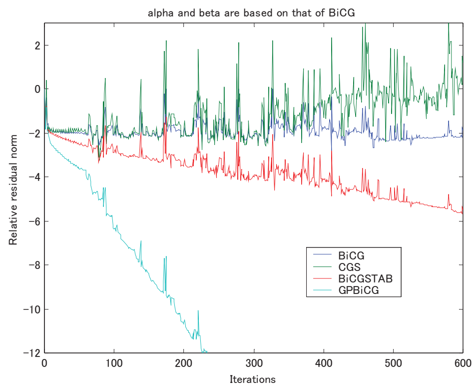


How Bi-CG part works?

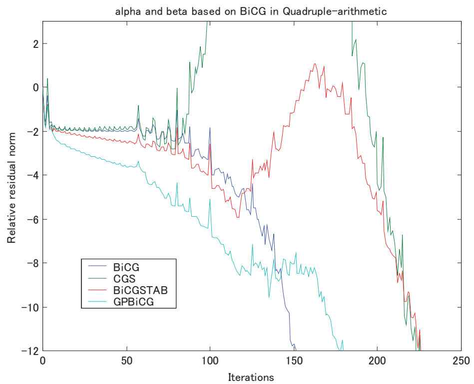
- Bi-CGSTAB converges by an effect of MR part (Bi-CG part is still unstable)
- GPBi-CG makes Bi-CG part stable
- CGS did not converge in Quadruple arithmetic
- In Quadruple arithmetic, simple Bi-CG is the best (Bi-CG is much affected by Rounding errors)
- In Quadruple arithmetic, Bi-CG part in Bi-CGSTAB is bad convergence even if Bi-CG converges.

Convergence history based on one Bi-CG

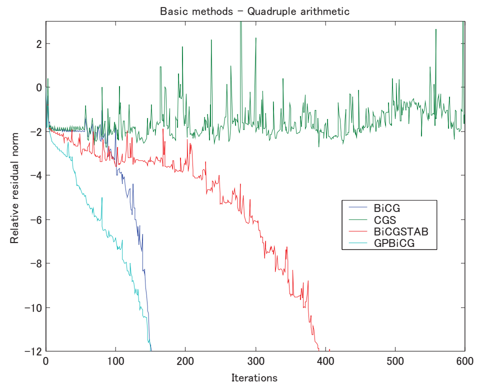
(alpha and beta in Bi-CG are used in all methods)



Convergence history based on one Bi-CG
(Quadruple arithmetic is used for Bi-CG)



Convergence history based on one Bi-CG
(Quadruple arithmetic is used for ALL)

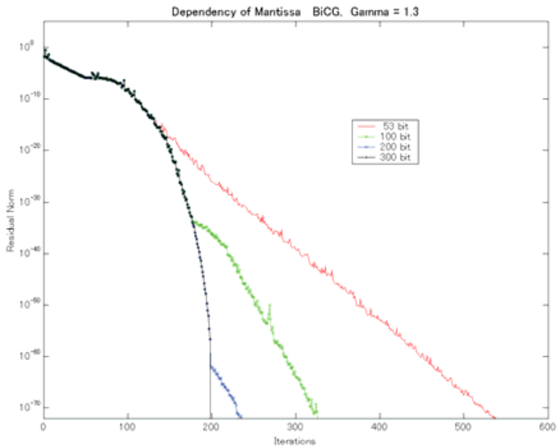


How accelerating polynomial works

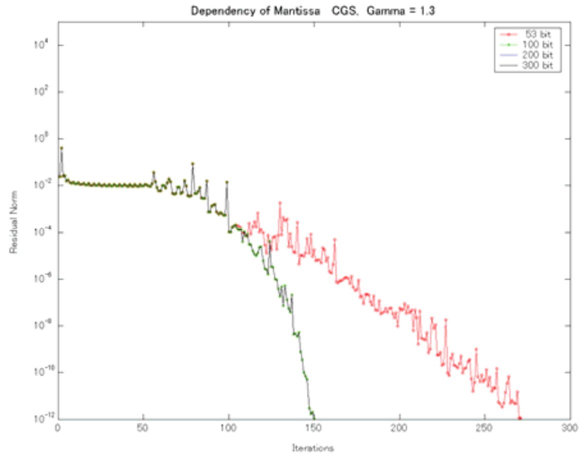
- Qudaruple arithmetic works very well.
- If enough accuracy was provided, Bi-CG was the best.
- Bi-CGSTAB and GPBi-CG work well.
- In Quadruple arithmetic, sometimes it works as braking not as accelerating.
- GPBi-CG is robust in both two conditions.
- CGS does not work in both conditions because of “squared”.

Utilizing Quadruple-Precision Floating Point Arithmetic Operation for the Krylov Subspace Methods

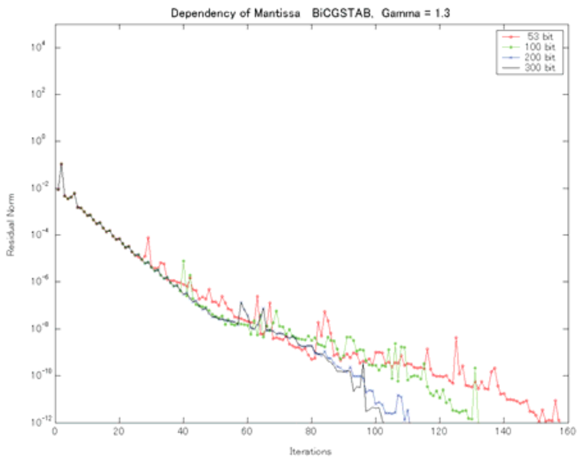
BiCG Gamma = 1.3



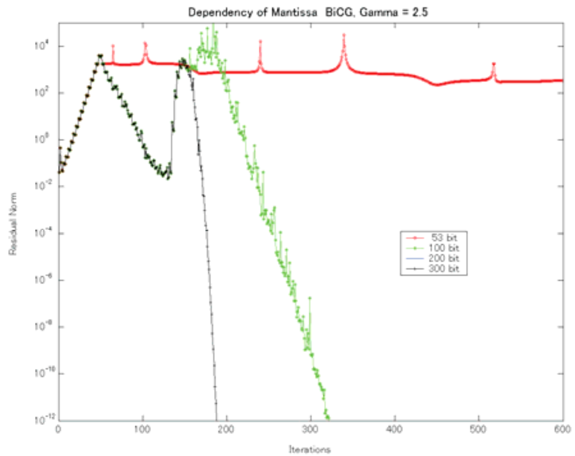
CGS Gamma = 1.3



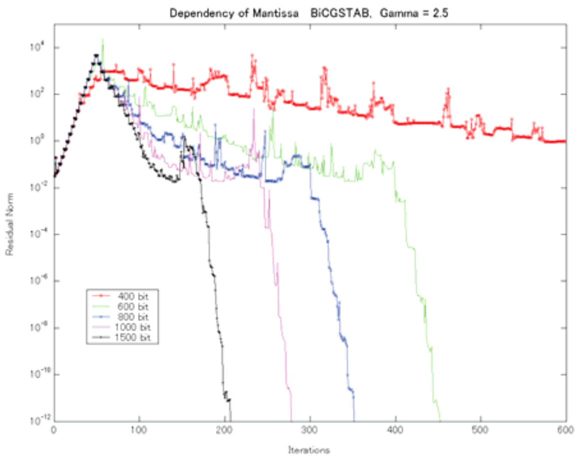
BiCGSTAB Gamma = 1.3



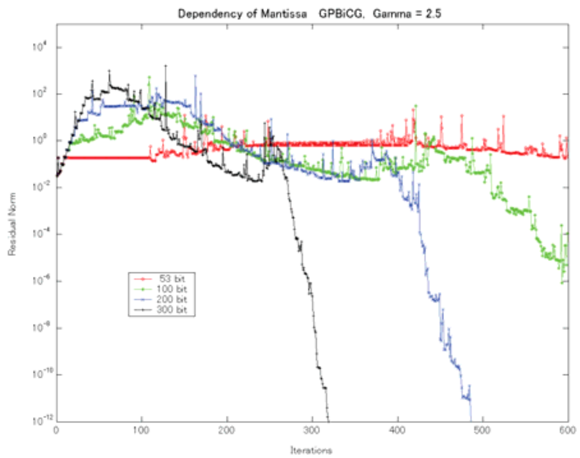
BiCG Gamma = 2.5



BiCGSTAB　Gamma = 2.5



GPBiCG　Gamma = 2.5



Observations

- Fast and smooth convergence are gained from More accurate computations.
- Required Mantissa is based on the problems:

BiCG	53 bit	for Gamma =	1.3
	100 bit	for	1.7
	200 bit	for	2.1
	200 bit	for	2.5

- Required Mantissa depends on Algorithms:

BiCG	200 bit and 190 iterations	
CGS	300 bit and 160	
x BiCGSTAB	1500 bit and 210	
x GPBiCG	300 bit and 310	(Gamma = 2.5)

High Precision Arithmetic

- Reducing round-off errors
- Accelerating algorithms ***mathematically***
- Not easy to use

High Precision Arithmetic without any Special Hardware

- Symbolic Computation (Computer Algebra)
- Variable length Multiple Precision
  - GMP
  - MBLAS
  - exflib
- Fixed length Multiple Precision
  - FORTRAN REAL *16
  - IEEE
  - Double-double

Important points!

- Full or Partial
  One Precision or Mixed Precisions
- Computing Environment
  Compiler/Emulation/Interpreter
- Program Interface, API

Our Solution:

Utilize Accurate Computations for Iterative Methods

- Use Double-double
- Use D-D vectors and Double Matrices
  (Mixed Precision Arithmetic Operations)
- Accelerate by SIMD
- Restart with different Precision
- Automatic Tuning
- Good tools

Advantages

- Tough for round-off errors
- Small Additional Memory
- Small Additional Communications
- Much Computations
- Applicable for **ALL** Iterative Methods
  (even if serial computation such as ILU)

Implementation of Fast Quadruple Arithmetic Operations

Double−Double(DD), Quad−Double(QD)

One DD number uses two double precision numbers.
One QD number uses four double precision numbers.

QD $$\mathrm{A} = a_0 + a_1 + a_2 + a_3$$

$$|a_{i+1}| \le \frac{1}{2}\mathrm{ulp}(a_i), \quad (i = 0, 1, 2)$$

※ulp (units in the last place)

Arithmetic operation is performed by using **normal** double precision operations.

D. H. Bailey, QD (C++ / Fortran−90 double−double and quad−double package), Available at http://crd.lbl.gov/~dhbailey/mpdist/

Implementation of Double-double Arithmetic

- Quadruple value is stored in two double floating point numbers
  - Double-double arithmetic: a = a.hi + a.lo, |a.hi|>|a.lo|
  - 8 bits less than IEEE standards
  - Effective digits are approx. 31 vs. 33 digits.

double-double arithmetic

	exponent 11 bit	Mantissa 52bit	+		Exponent 11bit	Mantissa 52 bit

IEEE Standards

	exponent 15bit	Mantissa 112bit

Round-off Error Free Double Arithmetic Addition

- Round-off error free addition can be done with two double precision variables:

$$a + b = \mathrm{fl}(a + b) + \mathrm{err}(a + b)$$

  - a,b : double floating point variables
  - fl(a + b) : addition of a and b in double
  - err(a+b) : (a+b) – fl(a+b): error

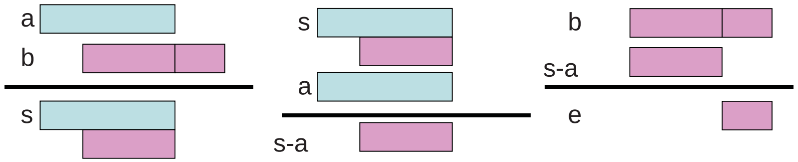
Basic Algorithm

- Dekker showed round-off error free addition in double

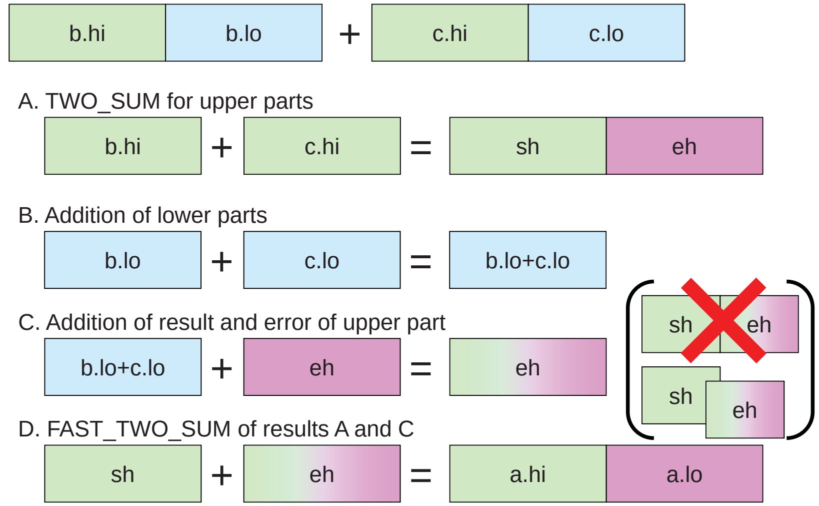
|a|>=|b| 3flops. Others 6flops.

```
FAST_TWO_SUM(a,b,s,e)
    s = a + b
    e = b - (s - a)
```

```
TWO_SUM(a,b,s,e)
    s = a + b
    v = s - a
    e = (a - (s - v)) + (b - v)
```



Quadruple Addition of a=b+c



Quadruple Addition of a=b+c

20 flops

```
ADD(a,b,c)
    TWO_SUM(b.hi,c.hi,sh,eh)
    TWO_SUM(b.lo,c.lo,sl,el)
    eh = eh + sl
    FAST_TWO_SUM(sh,eh,sh,eh)
    eh = eh + el
    FAST_TWO_SUM(sh,eh,a.hi,a.lo)
```

a=(a.hi,a.lo), b=(b.hi,b.lo), c=(c.hi,c.lo)

Number of operations

		+, −	*	/	total
DD	addition, subtraction	11	0	0	11
	multiplication	15	9	0	24
	division	17	8	2	27
QD	addition, subtraction	84	0	0	84
	multiplication	163	46	0	209
	division	713	88	5	806

*sloppy algorithm

Minimal Requirement

- +/-
- *
- /
- SQRT
- Input function
- Output function (print)
- Others

MuPAT

- MuPAT (Multiple Precision Arithmetic Toolbox) [2]
  - Double, DD, and QD as Scilab toolbox
  - Acceleration using C external functions

T. Saito, E. Ishiwata and H. Hasegawa, Development of quadruple precision arithmetic toolbox QuPAT on scilab, ICCSA2010, Proceedings Part II, (2010)
S. Kikkawa, T. Saito, E. Ishiwata and H. Hasegawa, Development and acceleration of multiple precision arithmetic toolbox MuPAT for Scilab , JSIAM Letters, Vol. 5, pp. 9–12 (2013)

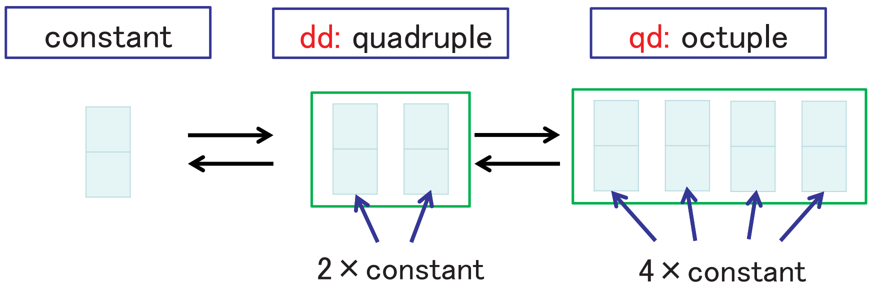
Ease of Use

- By trial and error
- No Programming
- Interactive
- Combination of D, DD, and QD
- Any machine

Scalar, Vector, and Matrix are treated as "constant" data type in Scilab

	constant		
	code	size	
Scalar	a = 1;	1×1	1
Vector	a = [1;2];	2×1	1 2
Matrix	a = [1,3;2,4];	2×2	1 3 2 4

Extension of Data types & Operators



Same operators {+, -, *, /} and functions should be defined between these data types.

"sparse" data type in Scilab

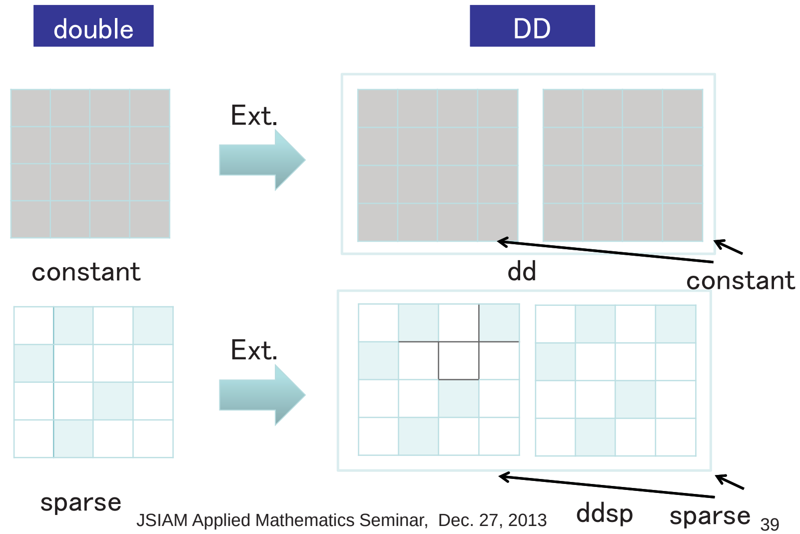
COOformat
(Coordinate list)

	2		5
1			
		4	
	3		

```
a =
(    4,    4) sparse
matrix
(    1,    2)        2.
(    1,    4)        5.
(    2,    1)        1.
(    3,    3)        4.
(    4,    2)        3.
   row  column       value
```

- "sparse" data type
  - Operators (+, −, ∗, /) are provided.
  - Reducing memory space:
    - ✓ # of Non zero is 5%→7.5% of constant data type

Extension of data types in two directions



Addition and Subtraction

Function name	operation
%sp_a_qd	sp + qd
%qd_a_sp	qd + sp
%qdsp_a_qdsp	qdsp + qdsp
%qdsp_a_ddsp	qdsp + ddsp
%ddsp_a_qdsp	ddsp + qdsp
%qdsp_a_sp	qdsp + sp
%sp_a_qdsp	sp + qdsp
%qdsp_a_qd	qdsp + qd
%qd_a_qdsp	qd + qdsp
%qdsp_a_dd	qdsp + dd
%dd_a_qdsp	dd + qdsp
%qdsp_a_s	qdsp + double
%s_a_qdsp	double + qdsp
%ddsp_a_qd	ddsp + qd
%qd_a_ddsp	qd + ddsp

- QD sparse $A+B$ 1000 times
  ($A,\ B$: qdsp, N =1000, # of Non zero 5%)

Scilab ・・・ 47.65 sec
C functions ・・・ 92.35 sec

⇨ Scilab only

- Same algorithms with dense

Multiplication

Function name	operation
%sp_m_qd	sp * qd
%qd_m_sp	qd * sp
%qdsp_m_qdsp	qdsp * qdsp
%qdsp_m_ddsp	qdsp * ddsp
%ddsp_m_qdsp	ddsp * qdsp
%qdsp_m_sp	qdsp * sp
%sp_m_qdsp	sp * qdsp
%qdsp_m_qd	qdsp * qd
%qd_m_qdsp	qd * qdsp
%qdsp_m_dd	qdsp * dd
%dd_m_qdsp	dd * qdsp
%qdsp_m_s	qdsp * double
%s_m_qdsp	double * qdsp
%ddsp_m_qd	ddsp * qd
%qd_m_ddsp	qd * ddsp

- DD sparse Ax 1,000 times

  (A : ddsp N=1,000,
  # of Non zero is 5%
  x : dd)

  Scilab・・・ 1135.78 sec
  C function・・・ 10.92 sec

  ⇒ Use C functions

- Sparse * Sparse = ? [4]

[4] Timothy A. Davis, Direct Methods for Sparse Linear Systems, SIAM, Philadelphia (2006).

Acceleration by C functions

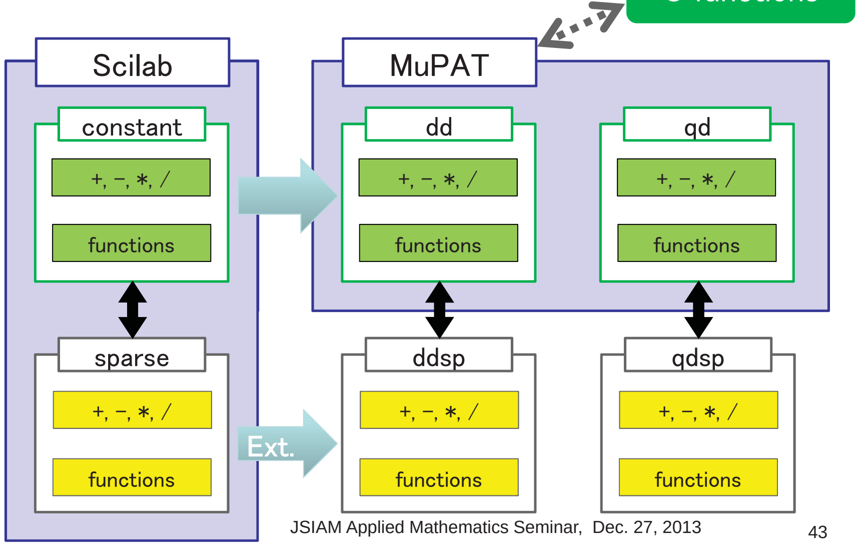
Repeated 10^6 times

Sec(ratio vs constant)

		+ / −	×	÷
d	# of double	1	1	1
	MuPAT	0.016	0.014	0.013
DD	# of double	11	24	27
	MuPAT	0.21 (12.8)	0.39 (28.4)	0.39 (30.6)
	MuPAT with C	0.26 (16.4)	0.31 (22.8)	0.32 (24.9)
QD	# of double	91	217	649
	MuPAT	2.91 (181.7)	4.21 (309.7)	21.29 (1663.5)
	MuPAT with C	0.34 (21.1)	0.39 (28.3)	0.39 (30.3)

※ Intel Core i5 2.5GHz, 4GB, Windows 7,Scilab version 5.3.3

Overview of new MuPAT



Memory Consumption

Matrix		Memory (MB)					
	Sparsity	constant	sparse	dd	ddsp	qd	qdsp
A	1%	8.00	0.12	16.00	0.25	32.00	0.41
B	5%	8.00	0.60	16.00	1.21	32.00	2.01
C	10%	8.00	1.21	16.00	2.41	32.00	4.01
D	66%	8.00	7.92	16.00	15.85	32.00	26.40
E	80%	8.00	9.60	16.00	19.21	32.00	32.01

Result of matrix operations (Memory)

	Sparsity	Memory (MB)	
		ddsp	qdsp
Ax	-	-	-
Bx	-	-	-
Cx	-	-	-
$A+B$	6%	1.43	2.39
$C+A$	11%	2.63	4.39
$B+C$	15%	3.49	5.82
AB	40%	9.51	15.98
CA	63%	15.17	25.17
BC	99%	23.86	39.73

Result of matrix operations (Computation Time, 100 times)

		Time (sec.)					
		dd	ddsp	dd/ddsp	qd	qdsp	qd/qdsp
	Ax	4.10	0.03	141.5	20.73	0.15	134.6
	Bx	4.13	0.14	30.2	20.76	0.74	28.0
	Cx	4.10	0.32	13.0	20.81	1.49	14.0
6%	$A+B$	6.36	0.64	10.0	15.97	1.16	13.7
11%	$C+A$	6.40	1.25	5.1	15.66	2.14	7.3
15%	$B+C$	6.35	1.69	3.7	15.85	2.90	5.5
40%	AB	2245.59	5.21	430.9	14909.50	12.27	1214.7
63%	CA	2288.42	8.10	282.6	14964.51	20.84	718.1
99%	BC	2282.17	16.54	138.0	14954.12	71.61	208.8

BiCG for ill-conditioned Problems

	Matrix				Time (sec.)		
		Iterations	Residual	Error	constant	sparse	c/s
D	west0497	†	1.02e+02	3.70e+05	39.1	2.8	13.8
	gre_1107	†	6.97e+03	1.69e+04	278.7	4.1	68.7
	tols2000	†	8.06e+02	2.34e+06	998.6	4.7	211.7
	sherman3	†	1.73e-03	6.24e-01	6749.1	11.7	577.6
	Matrix				Time (sec.)		
		Iterations	Residual	Error	dd	ddsp	dd/ddsp
DD	west0497	†	2.18e-01	7.73e+02	303.7	15.0	20.2
	gre_1107	†	2.40e-01	9.08e-01	1828.9	21.2	86.2
	tols2000	1586	9.29e-13	3.55e-09	938.3	4.1	228.7
	sherman3	7696	9.98e-13	1.05e-13	31227.4	45.8	681.9
	Matrix				Time (sec.)		
		Iterations	Residual	Error	qd	qdsp	qd/qdsp
QD	west0497	2676	6.09e-13	3.50e-08	306.6	7.0	43.84
	gre_1107	3401	8.59e-13	3.05e-11	2136.2	17.6	121.3
	tols2000	1080	6.77e-13	1.96e-09	2342.8	7.1	328.5
	sherman3	4884	9.35e-13	1.73e-13	–	91.1	

† : More than 10^4 iterations, – : Out of Memory

Lis & Lis-test

a Library of Iterative Solvers for linear systems

Lis has more than 10*13*11 combinations

Precond.
Jacobi
SSOR
ILU(k)
Hybrid
I+S
SAINV
SA-AMG
Crout ILU
additive schwarz
User defined

Solvers
CG
BiCG
CGS
BiCGSTAB
BiCGSTAB(l)
GPBiCG
BiCGSafe
Orthomin(m)
GMRES(m)
TFQMR
Jacobi
Gauss-Seidel
SOR

Storage Format
CRS: Compressed Row
CCS: Compressed Column
MSR: Modified Compressed Sparse Row
DIA: Diagonal
ELL: Ellpack-Itpack gen. diag.
JDS: Jagged Diagonal
COO: Coordinate
DNS: Dense
BSR: Block Sparse Row
BSC: Block Sparse Column
VBR: Variable Block Row

Steps

1. Initialize
2. Make matrix
3. Make vector
4. Define Solver
5. Set Values
6. Set conditions
7. Execute
8. Finalize

```
LIS_MATRIX    A;
LIS_VECTOR    b,x;
LIS_SOLVER    solver;
int           iter;
double        times,itimes,ptimes;

lis_initialize(argc, argv);
lis_matrix_create(LIS_COMM_WORLD,&A);
lis_vector_create(LIS_COMM_WORLD,&b);
lis_vector_create(LIS_COMM_WORLD,&x);
lis_solver_create(&solver);
lis_input(A,b,x,argv[1]);
lis_vector_set_all(1.0,b);
lis_solver_set_optionC(solver);
lis_solve(A,b,x,solver);
lis_solver_get_iters(solver,&iter);
lis_solver_get_times(solver,&times,
&itimes,&ptimes);
printf("iter = %d time = %e (p=%e
i=%e)¥n",iter,times, ptimes, itimes);
lis_finalize();
```

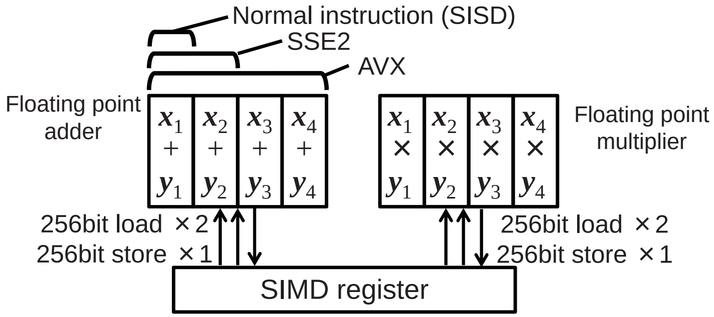

Design of Fast Quad. Operations for Lis

- Same API with Double
- Double: Input (A, b, x_0)
- Double: Output
- Double: Creation of Preconditioner M
- Fast Quad.: Iterative solution x
  All working variables
- Fast Quad.: Application of Preconditioner
  Mu=v

Acceleration

- SIMD is used for vectors(`dot`, `axpy`, `matvec`)
  - SSE2: 2 Multiply-and-add in same time
  - AVX: 4 Multiply-and-add in same time
  - AVX2: 4 Fused Multiply-and-add in same time
- 2 or 4 FMA in a loop with loop unrolling
  - pd instruction of SSE2 can be used for all
- Code tuning
  - Alignment
  - Some hand optimization

Architecture of intel core i7 2600k



- Addition and Multiplication in parallel
- Peak performance

  3.4G × 4 (AVX) × 2 (adder + multiplier) = 27.2 GFLOPS / core

  = 108.8 GFLOPS (4core)

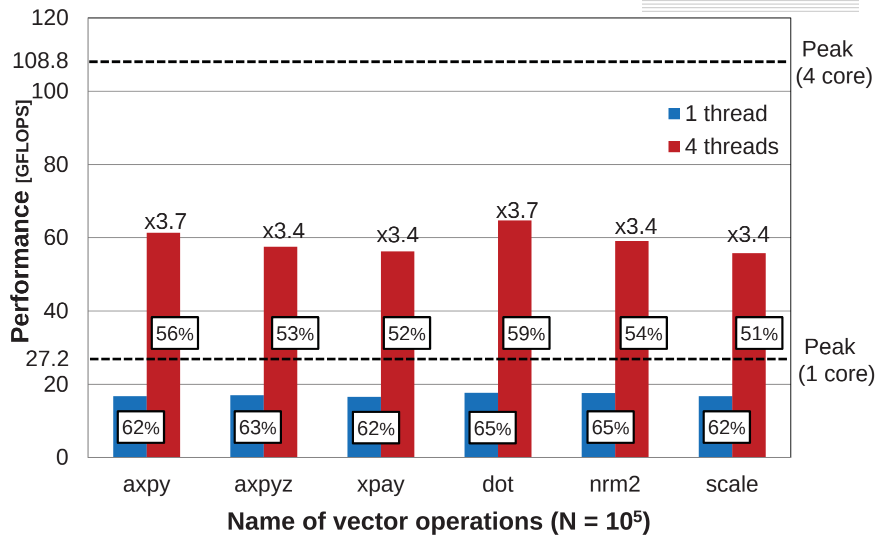
Vector operations (BLAS1)

	Operation	Memory access (Load + Store)	The number of double precision operations (add+sub : mult)
axpy	$\boldsymbol{y} = a\boldsymbol{x} + \boldsymbol{y}$	4 + 2	35 (26:9)
axpyz	$\boldsymbol{z} = a\boldsymbol{x} + \boldsymbol{y}$	4 + 2	35 (26:9)
xpay	$\boldsymbol{y} = \boldsymbol{x} + a\boldsymbol{y}$	4 + 2	35 (26:9)
dot	$val = \boldsymbol{x} \cdot \boldsymbol{y}$	4 + 0	35 (26:9)
nrm2	$val = \lVert\boldsymbol{x}^2\rVert$	2 + 0	31 (24:7)
scale	$\boldsymbol{x} = a\boldsymbol{x}$	2 + 2	24 (15:9)

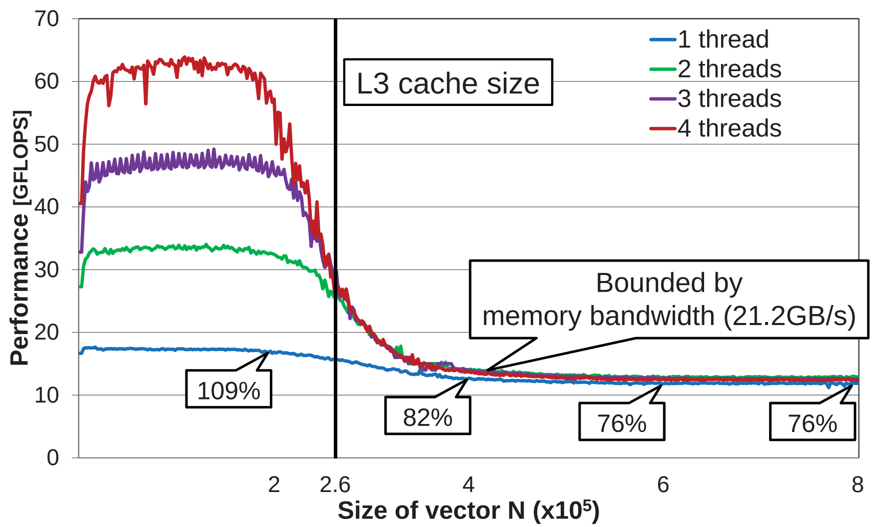
x, ***y***, ***z*** : DD vector; a, val : DD variable

"GFLOPS" := (# of double precision op. * N) / elapsed time

Performances of DD vector operations (in cache)



Performances of multi-threading (axpy)



Reducing memory access

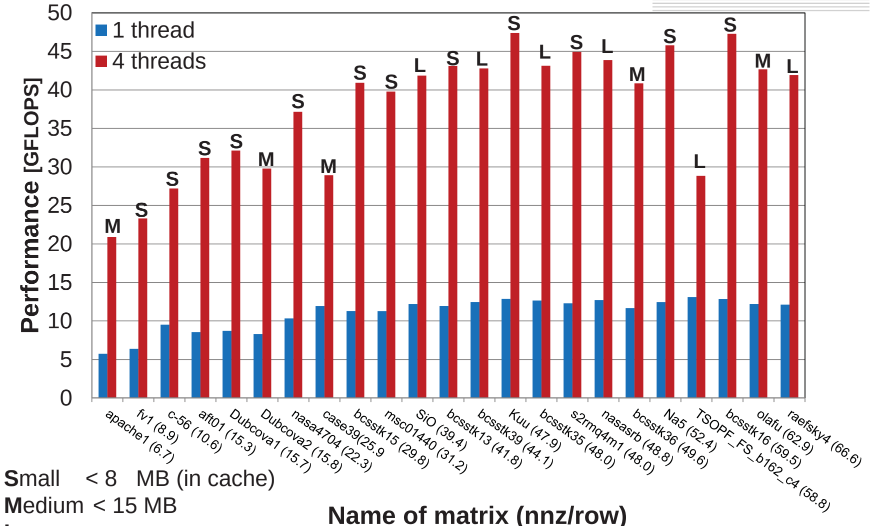
- CRS (compressed row storage) format is used

Bytes / flops of $\boldsymbol{y} = A\boldsymbol{x}$

	Bytes / flops
$\boldsymbol{y}_D = A_D\boldsymbol{x}_D$	14 (28 bytes / 2 flops)
$\boldsymbol{y}_{DD} = A_{DD}\boldsymbol{x}_{DD}$	1.5 (52 bytes / 35 flops)
◎ $\boldsymbol{y}_{DD} = A_D\boldsymbol{x}_{DD}$	1.3 (44 bytes / 33 flops)

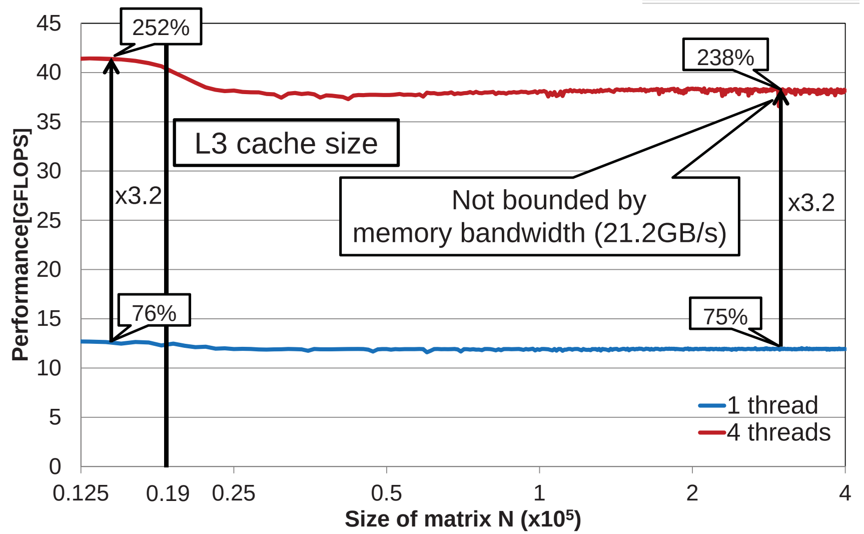
- $\boldsymbol{y}_{DD} = A_D\boldsymbol{x}_{DD}$
  - DD arithmetic is bounded by memory access
  - Input matrix A will be given by double precision
  - Data size is a half

Performance of SpMV

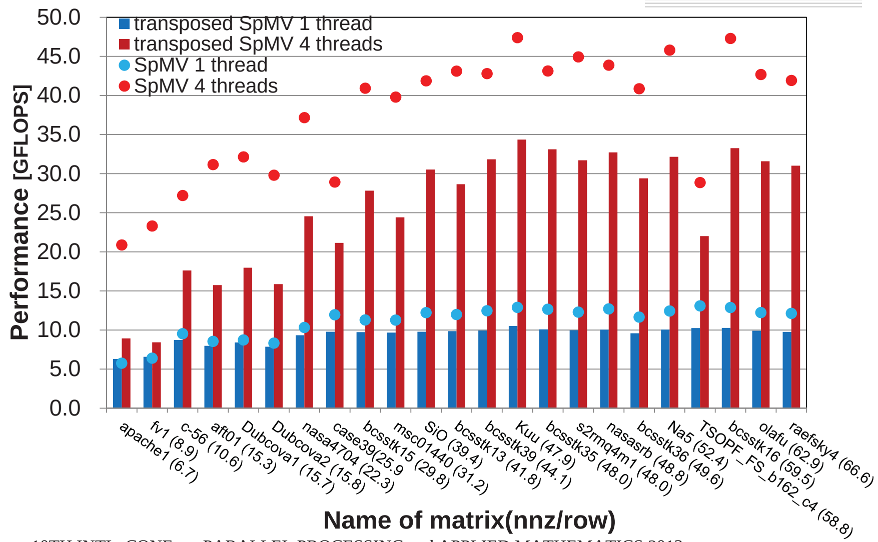


Small < 8 MB (in cache)
Medium < 15 MB
Large

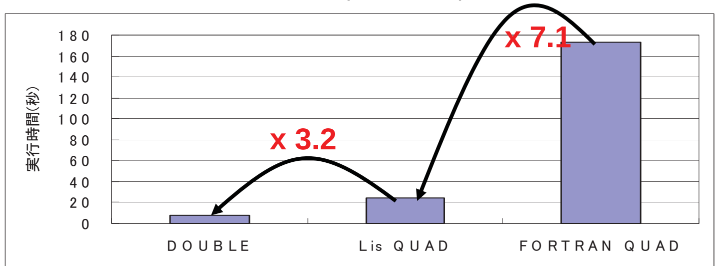
Performance of SpMV (bandmatrix (CRS), bandwidth = 32)



Performance of Transposed SpMV



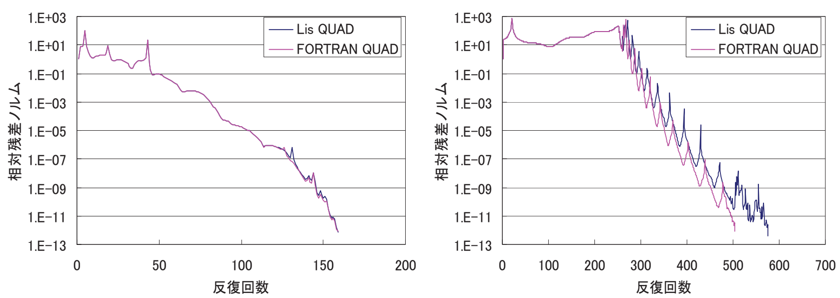
Time for 50 BiCG Iterations
Poisson（n=10^6, CRS）, Xeon 2.8GHz



	DOUBLE	Lis QUAD	FORTRAN QUAD
Matrix A(CRS)	4(n+nnz)+8nnz	4(n+nnz)+8nnz	4(n+nnz)+16nnz
Vector b	8n	8n	16n
Vector x	8n	16n	16n
Workings	6*8n	6*16n	6*16n
sum	121.9MB	175.8MB	221.6MB

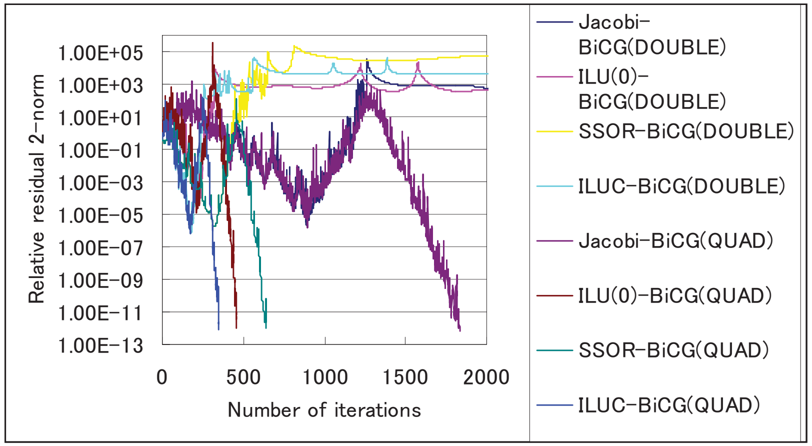
Comparison of Real *16 vs. Fast Quadruple with BiCG

rdb2048l (n=2048,cond=1.8E+3)　　olm1000 （n=1000,cond=3.0E+6)



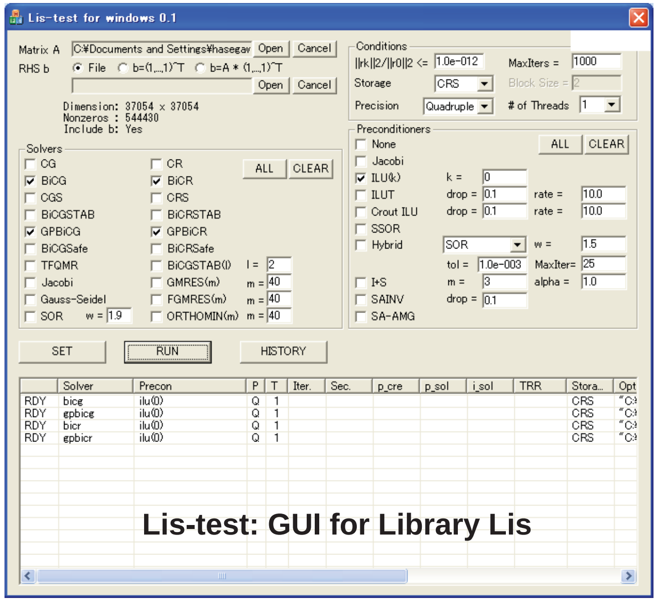
- Almost same accuracy (At most 10%)!

Convergence History of A4 with Preconditioned BiCG



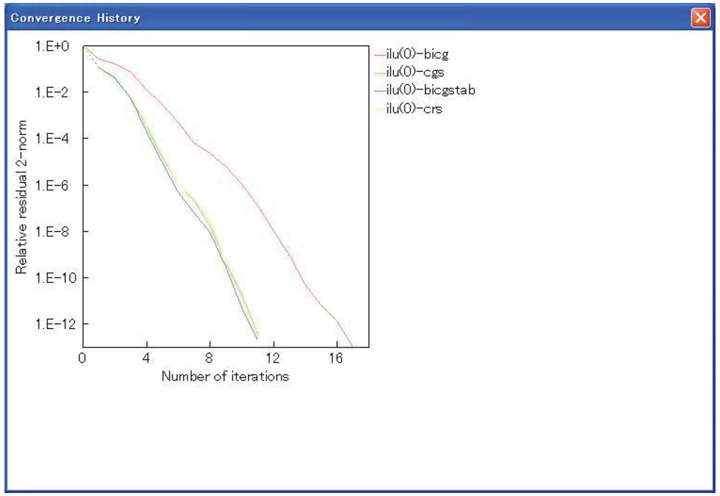
Lis-test for evaluation

- Over 2K combinations:
  10 Preconditioners x 13 Solvers x 11 Storage formats x 2 precisions
- Not necessary to install. Run from USB
- Prepare Matrix data as text file with Matrix Market' exchange format
- Run in parallel if the PC has multi-core
- To click, solutions, history, etc are computed



Lis-test: GUI for Library Lis

Comparison is done easily!



Algorithms

Basic Idea of Restart

- Until Now:

(1)Solve $Ax^{*}=b$ with some initial value x_0

(2)Solve $Ax=b$ with an initial value x^{*}

- – In general, (1) and (2) have same spaces, same methods, and same precisions
- – (1) and (2) have same spaces, same methods but **different precisions (combination of Double and Fast Quadruple)**.

SWITCH Algorithm

- Restart with different precision arithmetic
  - Current solution x_k is passed at the restart
  - Upper and Lower part of Double-Double var. are stored in different arrays
  - Only Upper part is used for Double Precision
  - Two Stages are performed by Different Precisions

```
for(k=0;k<matitr;k++){
    The first step
    if( nrm2<restart_tol ) break;
}
Clear all values except x
for(k=k+1;k<maxtr;k++) {
    The second step
    if( nrm2<tol ) break;
}
```


PERIODIC algorithm

- A Fast Quadruple is used each k iterations
  - All values are passed at the change
  - No cost at the change of Q → D
  - Lower part is cleared at the change of D → Q

```
for(k=0;k<maxitr;k++) {
  if( k%interval<num ) {
    Fast Quadruple is used
  } else {
    Lower part is cleared
    Double is used
  }
```

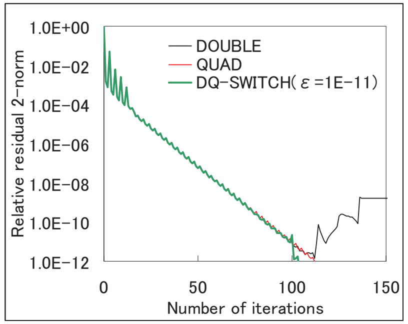

Teoplitzγ=1.3, n=10^5

		iter. total	double	sec.	‖b−Ax‖
FMA2_SSE2		113	0	6.60	2.47E−10
SWITCH	ε =1.0E−09	95	74	2.33	2.53E−10
	ε =1.0E−10	95	86	1.82	2.51E−10
	ε =1.0E−11	103	100	1.67	9.34E−11
PERIODIC	num=1	−	−		
	num=2	−	−		
	num=3	−	−		
	num=4	−	−		
	num=5	107	52	3.98	3.16E−10
	num=6	118	46	4.89	2.02E−10

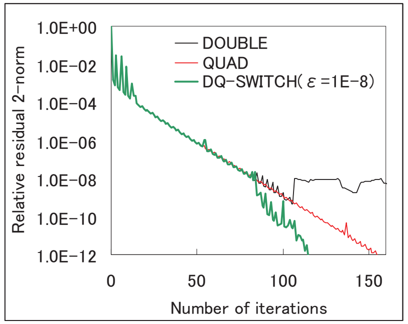
- Epsilon is restart criterion of DQ-SWITCH
- Num: Quad. Ops. Used num times per 10 iterations

Convergence History

γ=1.3



γ=1.4



- DQ-SWITCH is good convergence

Epsilon dependency

	ε	γ=1.3 iter. total	double	quad	sec	γ=1.4 iter. total	double	quad	sec
QUAD		113			2.88	155			3.94
DQ-SWITCH	1.00E-03	114	2	112	2.87	156	2	154	3.94
	1.00E-04	109	11	98	2.59	152	15	137	3.62
	1.00E-05	105	23	82	2.26	146	31	115	3.16
	1.00E-06	104	35	69	2.01	138	47	91	2.67
	1.00E-07	95	47	48	1.58	123	65	58	1.96
	1.00E-08	94	61	33	1.29	119	83	36	1.53
	1.00E-09	95	74	21	1.08	--			
	1.00E-10	95	86	9	0.86	--			
	1.00E-11	103	98	5	0.84	--			

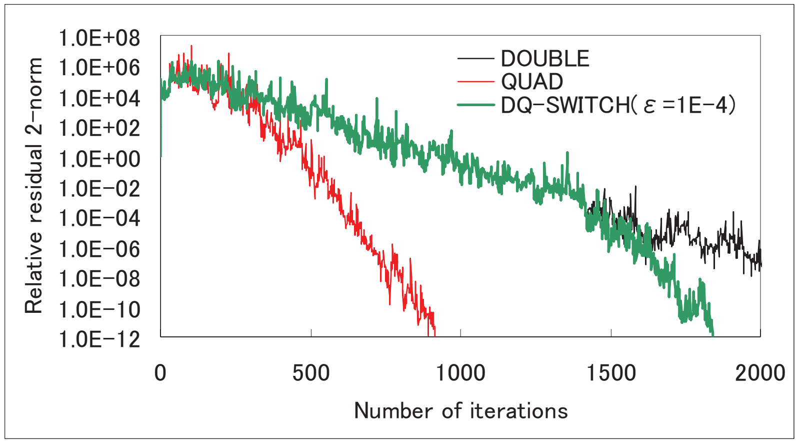
- Choice of appropriate epsilon is important
- Small epsilon reduces much computation time
- Smaller epsilon makes divergence

airfoil_2d		iter. total	double	sec.	||b−Ax||
DOUBLE		4567	4567	18.64	3.25E−08
QUAD		3838		69.39	5.36E−10
SWITCH	ε =1.0E−10	4402	4091	24.25	3.15E−10
	ε =1.0E−11	4331	4176	21.66	3.13E−10
	ε =1.0E−12	4709	4567	22.87	3.56E−10
wang3		**total**	**double**	**sec.**	**||b−Ax||**
DOUBLE		476	476	2.03	3.52E−10
QUAD		372		7.31	1.49E−10
SWITCH	ε =1.0E−10	460	361	3.67	1.59E−10
	ε =1.0E−11	459	444	2.42	9.22E−11
	ε =1.0E−12	479	476	2.32	1.46E−10
language		**total**	**double**	**sec.**	**||b−Ax||**
DOUBLE		39	39	3.42	2.96E−09
QUAD		36		10.53	4.25E−11
SWITCH	ε =1.0E−10	38	34	4.57	1.71E−10
	ε =1.0E−11	37	35	4.07	4.20E−10
	ε =1.0E−12	40	39	4.18	4.47E−10

- ε is restaring criterion of SWITCH

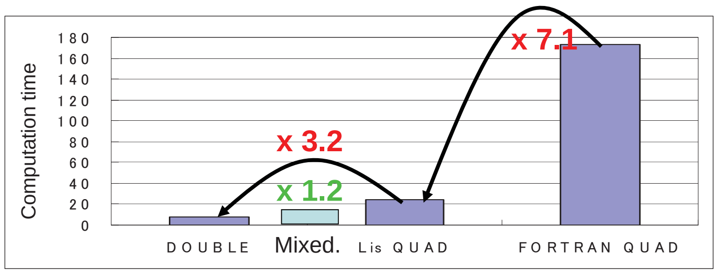
- QUAD and SWITCH improve 2 digits for solution' quality
- SWITCH is 20% overhead on the double, however robust

A4: Electronics Effect



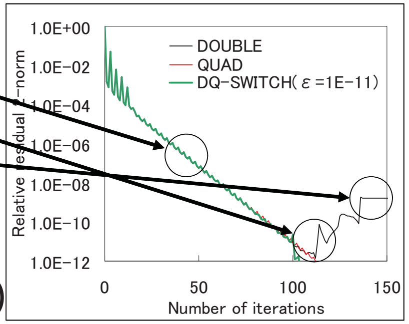
Computation Time

Poisson（n=10^6, CRS）, Xeon 2.8GHz



Auto Restart with Different Precisions

- Convergent history shows three patterns:
  - (C)Converge
  - (D)Diverge
  - (S)Stagnate



- To Detect (D) and (S), restart at the point

Auto Restart of DQ-SWITCH

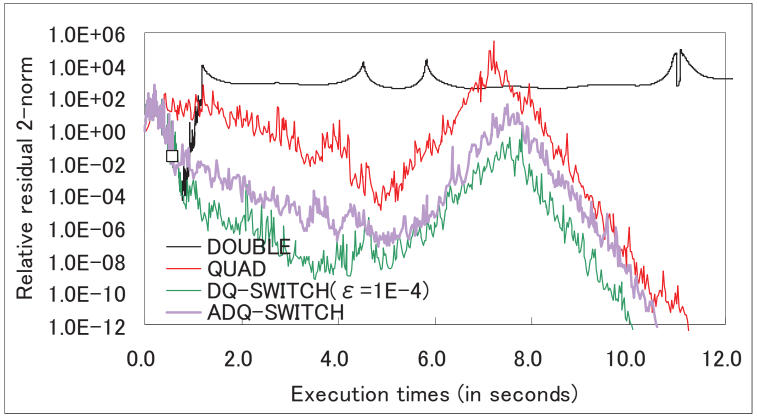
- Compute deviation of residual norm

$$v = \frac{1}{p}\sum_{i=1}^{p}\left(\frac{nrm(i)-nrm(1)}{nrm(1)}\right)^2$$

- (D) $v \geq 10^2$
- (S) $v \leq 10^{-1}$

```
if( nrm2 < nrm2_min )
   nrm2_min = nrm2;  x_bak = x;
nrm_bak[k%10] = nrm2;
if( k>=10 ) {
  v = 0.0;   c = 0;
  for(i=0;i<10;i++) {
    t = nrm_bak[i] - nrm_bak[(k-9)%10];
    t = t / nrm_bak[(k-9)%10];
    v = v + t*t;
    if( nrm_bak[(k-9)%10] <= nrm_bak[i] )
      c = c+1;
  }
  v = v / 10;
  if( v<=0.1 || (c==10 && v>=100) ) break;
  if( nrm2<tol ) break;
}
```


Electoronics BiCG with ILU(0)



- Divergence or Stagnation is detected.
- Computation time is reduced.

Mixed Precision Iterative Methods

- Complicated problems are solved with Mixed or QUAD.
- Overhead of the mixed precision iterative methods is 20%
- SWITCH is Good at least 2 digits with 20% more
  - D → Q: easy, robust, however depends on timing of restart
- Auto restart of DQ-SWITH
  - Deviation is used to detect “Diverge” or “Stagnate”

Parallel Issues for Fast Quad.

- Depends on the implementation of Ax, $A^T x$, $M^{-1}x$, $M^{-T}x$, and Matrix Storage Format
- Data transfered is almost same
- Heavy Computation

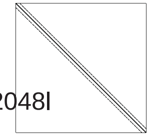
→ Suitable for Distributed Parallel

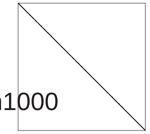
- Less round-off errors

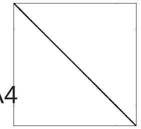
→ lighter preconditioner (easy to parallelize)

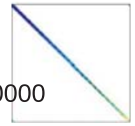
Test Problems

- Poisson
  - 2 dimension, FDM
  - N=10^6, nnz=5x10^6
- rdb2048l (Chemical engineering)
  - MatrixMarket, n=2048, nnz=12032, cond = 1.8x10^3
- olm1000 (Hydrodynamics)
  - MatrixMarket, n=1000, nnz=3996, cond = 3x10^6
- A4 (Electronic potential)
  - n=23,994, nnz=214,060
- **Cryg10000 (CRYSTAL GROWTH EIGENMODES)**
  - UF Sparse Matrix Collection, n=10000, nnz=49699
- circuit_3 (Circuit Simulation)
  - n=12,127, nnz=48,137


rdb2048l


olm1000


A4


cryg10000

Comparison of Double and DQ-SWITCH

- University of Florida Sparse Matrix Collection

Matrix	dimension	nnz	Size of memory	
			Double	Lis Quad
airfoil_2d	14,214	259,688	3.9MB	4.7MB
wang3	26,064	177,168	3.7MB	5.1MB
language	399,130	1,216,334	39.8MB	61.1MB

Application Program Interface

- Data Types (Precision)
- Matrix Storage Format
- Algorithms
- Function names
- Computing Environments

SILC

The traditional way of using libraries

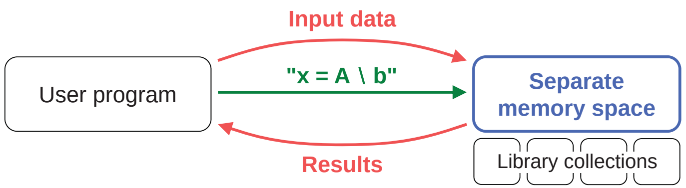
1. Preparation of matrices and vectors using library-specific data structures
2. Function calls with a function's name and its arguments in a prescribed order

As a result...

- User programs will depend on a specific library
  - Not easy to replace the library by another

SILC: Simple Interface for Library Collections

- Basic ideas
  - **Data transfer** and **a request for computation**
  - **Mathematical expressions** for the request
  - **A separate memory space** for the computation



Solving a system of linear equations $Ax=b$

- In the traditional way (using LAPACK in C)

```
double *A, *b;
int kl, ku, lda, ldb, nrhs, info, *ipiv;

dgbtrf (N, N, kl, ku, A, lda, ipiv, &info);  /* LU factorization */
if (info == 0)
   dgbtrs ('N', N, kl, ku, nrhs, A, lda, ipiv, b, ldb, &info);  /* solve */
```

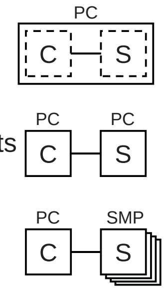
- In SILC

```
silc_envelope_t A, b, x;

SILC_PUT ("A", &A);
SILC_PUT ("b", &b);
SILC_EXEC ("x = A \ b");  /* call a solver (e.g., dgbtrf & dgbtrs) */
SILC_GET (&x, "x");
```

Main benefits of using SILC

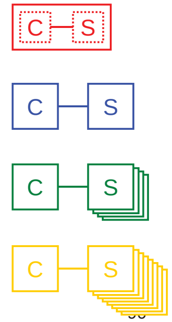
- Source-level independence between user programs and matrix computation libraries
  - Easy access to alternative solvers and matrix storage formats, possibly in other libraries
  - Instant porting to other computing environments without any modification in user programs
- You need to prepare only the smallest amount of data
  - Temporary buffers are automatically allocated
- Language-independent mathematical expressions
  - Applicable in many programming languages (C, Fortran, Python, MATLAB)



SILC servers in different computing environments

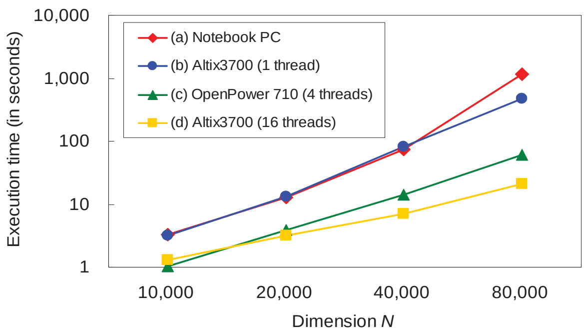
- A user program (client) that solves $Ax = b$
  - Where A is a tridiagonal matrix in the CRS format
  - Run in the notebook PC of Environment (a)
  - In a 100-Base TX local-area network

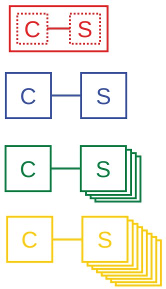
	Environment	Specification	OpenMP
(a)	A notebook PC	Intel Pentium M 733 1.1GHz, 768MB memory, Fedora Core 3	N/A
(b)	SGI Altix3700	Intel Itanium2 1.3GHz × 32, 32GB memory, Red Hat Linux Advanced Server 2.1	1 thread
(c)	IBM eServer OpenPower 710	IBM Power5 1.65GHz × 2 (4 logical CPUs), 1GB memory, SuSE Linux Enterprise Server 9	4 threads
(d)	SGI Altix3700	Same as (b)	16 threads



Experimental results

- About 0.1 second of data communications over the LAN
  - Data size: 0.46MB (N=10,000) to 4.27MB (N=80,000)
- SILC servers in (c) and (d) achieved better performance because of parallel computation





Functionalities

- Data structures
  - Data types: scalar, vector, matrix, cubic array
  - Precisions: integer, real, complex (single or double)
  - Matrix storage formats: dense, banded, CRS
- Mathematical expressions
  - Binary arithmetic operators (+, −, *, /, %)
  - Solutions of systems of linear equations (A \ b)
  - Conjugate transposes (A'), complex conjugates (A~)
  - Built-in functions
    - Ex. "sqrt(b' * b)" is the 2-norm of vector b
  - Subscript
    - Ex. "A[1:5,1:5]" is a 5×5 submatrix of A

Conclusion

- Accurate Computation
  - Powerful tool for "Iterative Methods"
  - Another choice for designing Algorithm
  - Tool for analysis
- MuPAT: Ease of Use of D-D and Q-D
- Lis: Iterative Solvers with Fast D-D
- Lis-test: the simplest tool
- SILC: General Purpose API

Collaborators and Acknowledgement

- Lis & Lis-test
  H. Kotakemori
  A. Fujii (Kogakuin U)
  K. Nakajima (U Tokyo)
  A. Nishida(U Kyushu)
- Acceleration on AVX
  T. Hishinuma (Kogakuin U.)
  K. Asakawa (Kogakuin U)
  A. Fujii (Kogakuin U)
  T. Tanaka (Kogakuin U)
- SILC
  T. Kajiyama (Universidade Nova de Lisboa)
  A. Nukada (TITECH)
  R. Suda (U Tokyo)
  A. Nishida (U Kyushu)
- MuPAT
  T. Saitoh (Tokyo U of Science)
  S. Kikkawa (TUS)
  E. Ishiwata (TUS)

Lis and SILC are parts of SSI project which is funded by JST/CREST

Appendix A : Algorithms for DD and QD Arithmetics

We describe the details of the algorithms for DD and QD arithmetics. The procedures of algorithms are based on Knuth [5], Dekker [3], Priest [9], Shewchuk [14], Bailey [1] and Hida et al. [4].

A.1 Preliminaries for DD and QD arithmetics

In this section, we introduce some algorithms of floating-point arithmetic.

Assuming that $|a| \geq |b|$, Algorithm A.1, Fast-Two-Sum, produces a nonoverlapping expansion $s+e$ such that $a+b=s+e$, where s is an approximation to $a+b$ and e represents the round-off error in the calculation of s, in [14, p. 312].

Algorithm A.2, Two-Sum, is similar to Algorithm A.1, but Algorithm A.2 does not require the condition of $|a| \geq |b|$.

Algorithm A.1 Fast-Two-Sum(a,b) : Assume that $|a| \geq |b|$

1: $s \leftarrow a \oplus b$
2: $v \leftarrow s \ominus a$
3: $e \leftarrow b \ominus v$
4: **return** (s,e)

Algorithm A.2 Two-Sum(a,b)

1: $s \leftarrow a \oplus b$
2: $v \leftarrow s \ominus a$
3: $e \leftarrow (a \ominus (s \ominus v)) \oplus (b \ominus v)$
4: **return** (s,e)

Algorithm A.3, Split, produces a 26 bit value a_h and a nonoverlapping 26 bit value a_l such that $|a_h| > |a_l|$ and $a = a_h + a_l$, in [14, p. 325].

Algorithm A.3 Split(a)

1: $t \leftarrow 134217729 \otimes a$
2: $v \leftarrow t \ominus a$
3: $a_h \leftarrow t \ominus v$
4: $a_l \leftarrow a \ominus a_h$
5: **return** (a_h, a_l)

Algorithm A.4, Two-Prod, produces a nonoverlapping expansion $p+e$ such that $a \times b = p + e$, where p is an approximation to $a \times b$ and e represents the round-off error in the calculation of p, in [14, p. 326].

Algorithm A.4 Two-Prod(a,b)

1: $p \leftarrow a \otimes b$
2: $[a_h, a_l] \leftarrow$Split(a)
3: $[b_h, b_l] \leftarrow$Split(b)
4: $e \leftarrow ((a_h \otimes b_h \ominus p) \oplus a_h \otimes b_l \oplus a_l \otimes b_h) \oplus a_l \otimes b_l$
5: **return** (p,e)

Algorithm A.5, Two-Sqr, produces a nonoverlapping expansion $p+e$ such that $a^2 = p+e$, where p is an approximation to a^2 and e represents the round-off error in the calculation of p.

Algorithm A.5 Two-Sqr(a)

1: $p \leftarrow a \otimes a$
2: $[a_h, a_l] \leftarrow$Split(a)
3: $e \leftarrow ((a_h \otimes a_h \ominus p) \oplus (a_h \otimes a_l) \otimes 2.0) \oplus a_l \otimes a_l$
4: **return** (p,e)

Algorithm A.6, Three-Sum, produces a nonoverlapping expansion $d+e+f$ such that $a+b+c = d+e+f$, in [4].

Algorithm A.6 Three-Sum(a,b,c)

1: $[t_0, t_1] \leftarrow$ Two-Sum(a,b)
2: $[d, t_2] \leftarrow$ Two-Sum(t_0, c)
3: $[e, f] \leftarrow$ Two-Sum(t_1, t_2)
4: **return** (d,e,f)

Algorithm A.7, Three-Sum2, produces two double-precision numbers $d = (a \oplus b) \oplus c$ and $e = (a+b+c) - s$, in [4].

Algorithm A.7 Three-Sum2(a,b,c)

1: $[t_0, t_1] \leftarrow$ Two-Sum(a,b)
2: $[d, t_2] \leftarrow$ Two-Sum(t_0, c)
3: $e = t_1 \oplus t_2$
4: **return** (d,e)

Supposing that a_0, a_1, a_2, a_3 and a_4 construct a five-term expansion with limited overlapping bits, with a_0 being the most significant component. Then Algorithm A.8, Renormalize, produces a four-term nonoverlapping expansion $b_{(qd)} = b_0 + b_1 + b_2 + b_3$.

Algorithm A.8 Renormalize(a_0, a_1, a_2, a_3, a_4)

$[s, t_3] \leftarrow$ Fast-Two-Sum(a_3, a_4)
$[s, t_2] \leftarrow$ Fast-Two-Sum(a_2, s)
$[s, t_1] \leftarrow$ Fast-Two-Sum(a_1, s)
$[b_0, t_0] \leftarrow$ Fast-Two-Sum(a_0, s)
$[s, t_2] \leftarrow$ Fast-Two-Sum(t_2, t_3)
$[s, t_1] \leftarrow$ Fast-Two-Sum(t_1, s)
$[b_1, t_0] \leftarrow$ Fast-Two-Sum(t_0, s)
$[s, t_1] \leftarrow$ Fast-Two-Sum(t_1, t_2)
$[b_2, t_0] \leftarrow$ Fast-Two-Sum(t_0, s)
$b_3 = t_0 \oplus t_1$
return (b_0, b_1, b_2, b_3)

Algorithm A.9, Renormalize2, produces a four-term nonoverlapping expansion $b_{(qd)} = b_0 + b_1 + b_2 + b_3$. This algorithm is similar to Algorithm A.8 except for the number of arguments.

Algorithm A.9 Renormalize2(a_0, a_1, a_2, a_3)

$[s, t_2] \leftarrow$ Fast-Two-Sum(a_2, a_3)
$[s, t_1] \leftarrow$ Fast-Two-Sum(a_1, s)
$[b_0, t_0] \leftarrow$ Fast-Two-Sum(a_0, s)
$[s, t_1] \leftarrow$ Fast-Two-Sum(t_1, t_2)
$[b_1, t_0] \leftarrow$ Fast-Two-Sum(t_0, s)
$[b_2, b_3] \leftarrow$ Fast-Two-Sum(t_0, t_1)
return (b_0, b_1, b_2, b_3)

Table 11 shows the number of double-precision arithmetic operations for Algorithm A.1 $\sim$ A.9.

Table 11: Number of double-precision arithmetic operations for Algorithm A.1 $\sim$ A.9

Algorithm	$\oplus$, $\ominus$	$\otimes$	Total
Fast-Two-Sum (A.1)	3	0	3
Two-Sum (A.2)	6	0	6
Split (A.3)	3	1	4
Two-Prod (A.4)	10	7	17
Two-Sqr (A.5)	7	5	12
Three-Sum (A.6)	18	0	18
Three-Sum2 (A.7)	13	0	13
Renormalize (A.8)	28	0	28
Renormalize2 (A.9)	18	0	18

A.2 Algorithms for DD arithmetic

In this section, we show the algorithms of four arithmetic operations for double-double numbers.

A.2.1 addition

Algorithm A.10, dd_d_add, shows the procedure for adding a double-precision number b to a double-double number $a_{(dd)}$ and returns the double-double number $c_{(dd)} = c_0 + c_1$. If you want to add a double-double number $b_{(dd)}$ to a double-precision number a, dd_d_add (b_0, b_1, a) returns the result. d_dd_add is same as dd_d_add.

Algorithm A.10 dd_d_add (a_0, a_1, b)

$[s, e] \leftarrow$ Two-Sum(a_0, b)
$e \leftarrow e \oplus a_1$
$[c_0, c_1] \leftarrow$ Fast-Two-Sum(s, e)
return (c_0, c_1)

Algorithm A.11, dd_dd_add, shows the procedure for adding a double-double number $b_{(dd)}$ to a double-double number $a_{(dd)}$ and returns the double-double number $c_{(dd)} = c_0 + c_1$.

Algorithm A.11 dd_dd_add (a_0, a_1, b_0, b_1)

$[s, e] \leftarrow$ Two-Sum(a_0, b_0)
$e \leftarrow e \oplus (a_1 \oplus b_1)$
$[c_0, c_1] \leftarrow$ Fast-Two-Sum(s, e)
return (c_0, c_1)

A.2.2 subtraction

Algorithm A.12 (A.13), dd_d_sub (d_dd_sub), shows the procedure for subtracting a double- precision number b (a double-double number $b_{(dd)}$) from a double-double number $a_{(dd)}$ (a double-precision number a) and returns the double-double number $c_{(dd)} = c_0 + c_1$.

Algorithm A.12 dd_d_sub (a_0, a_1, b)

$s \leftarrow -b$
$[c_0, c_1] \leftarrow$ dd_d_add (a_0, a_1, s)
return (c_0, c_1)

Algorithm A.13 d_dd_sub (a, b_0, b_1)

$s_0 \leftarrow -b_0$
$s_1 \leftarrow -b_1$
$[c_0, c_1] \leftarrow$ d_dd_add (a, s_0, s_1)
return (c_0, c_1)

Algorithm A.14, dd_dd_sub, shows the procedure for subtracting a double-double number $b_{(dd)}$ from a double-double number $a_{(dd)}$ and returns the double-double number $c_{(dd)} = c_0 + c_1$.

Algorithm A.14 dd_dd_sub (a_0, a_1, b_0, b_1)

$s_0 \leftarrow -b_0$
$s_1 \leftarrow -b_1$
$[c_0, c_1] \leftarrow$ dd_dd_add (a_0, a_1, s_0, s_1)
return (c_0, c_1)

A.2.3 multiplication

Algorithm A.15, dd_d_mul, shows the procedure for multiplying a double-double number $a_{(dd)}$ by a double-precision number b and returns the double-double number $c_{(dd)} = c_0 + c_1$. d_dd_mul is same as dd_d_mul.

Algorithm A.15 dd_d_mul (a_0, a_1, b)

$[p, e] \leftarrow$ Two-Prod(a_0, b)
$e \leftarrow e \oplus (a_1 \otimes b)$
$[c_0, c_1] \leftarrow$ Fast-Two-Sum(p, e)
return (c_0, c_1)

Algorithm A.16, dd_dd_mul, shows the procedure for multiplying a double-double number $a_{(dd)}$ by a double-double number $b_{(dd)}$ and returns the double-double number $c_{(dd)} = c_0 + c_1$.

Algorithm A.16 dd_dd_mul (a_0, a_1, b_0, b_1)

$[p, e] \leftarrow$ Two-Prod(a_0, b_0)
$e \leftarrow e \oplus (a_0 \otimes b_1)$
$e \leftarrow e \oplus (a_1 \otimes b_0)$
$[c_0, c_1] \leftarrow$ Fast-Two-Sum(p, e)
return (c_0, c_1)

A.2.4 division

Supposing that $b \neq 0$ and $b_0 \neq 0$. Algorithm A.17 (A.18), dd_d_div (d_dd_div), shows the procedure for dividing a double-double number $a_{(dd)}$ (a double-precision number a) by a double-precision number b (a double-double number $b_{(dd)}$) and returns the double-double number $c_{(dd)} = c_0 + c_1$.

Algorithm A.17 dd_d_div (a_0, a_1, b)

$c_0 \leftarrow a_0 \oslash b$
$[p, e] \leftarrow$ Two-Prod(c_0, b)
$c_1 \leftarrow ((a_0 \ominus p) \ominus e \oplus a_1) \oslash b$
$[c_0, c_1] \leftarrow$ Fast-Two-Sum(c_0, c_1)
return (c_0, c_1)

Algorithm A.18 d_dd_div (a, b_0, b_1)

$c_0 \leftarrow a \oslash b_0$
$[p, e] \leftarrow$ Two-Prod(c_0, b_0)
$c_1 \leftarrow ((a \ominus p) \ominus e \ominus c \otimes b_1) \oslash b_0$
$[c_0, c_1] \leftarrow$ Fast-Two-Sum(c_0, c_1)
return (c_0, c_1)

Supposing that $b_0 \neq 0$. Algorithm A.19, dd_dd_div, shows the procedure for dividing a double-double number $a_{(dd)}$ by a double-double number $b_{(dd)}$ and returns the double-double number $c_{(dd)} = c_0 + c_1$.

Algorithm A.19 dd_dd_div (a_0, a_1, b_0, b_1)

$c_0 \leftarrow a_0 \oslash b_0$
$[p, e] \leftarrow$ Two-Prod(c_0, b_0)
$c_1 \leftarrow ((a_0 \ominus p) \ominus e \oplus a_1 \ominus c \otimes b_1) \oslash b_0$
$[c_0, c_1] \leftarrow$ Fast-Two-Sum(c_0, c_1)
return (c_0, c_1)

Table 12 shows the number of double precision arithmetic operations for double-double arithmetic.

A.3 Algorithms for QD arithmetic

A.3.1 addition

Algorithm A.20, qd_d_add, shows the procedure for adding a double-precision number b to a quad-double number $a_{(qd)}$ and returns the quad-double number $c_{(qd)} = c_0 + c_1 + c_2 + c_3$.

Algorithm A.20 qd_d_add (a_0, a_1, a_2, a_3, b)

$[c_0, e] \leftarrow$ Two-Sum(a_0, b)
$[c_1, e] \leftarrow$ Two-Sum(a_1, e)
$[c_2, e] \leftarrow$ Two-Sum(a_2, e)
$[c_3, e] \leftarrow$ Two-Sum(a_3, e)
$[c_0, c_1, c_2, c_3] \leftarrow$ Renormalize(c_0, c_1, c_2, c_3, e)
return (c_0, c_1, c_2, c_3)

Algorithm A.21, qd_dd_add, shows the procedure for adding a double-double number $b_{(dd)}$ to a quad-double number $a_{(qd)}$ and returns the quad-double number $c_{(qd)} = c_0 + c_1 + c_2 + c_3$.

Table 12: Number of double precision arithmetic operations for double-double arithmetic

	Algorithm	$\oplus$ & $\ominus$	$\otimes$	$\oslash$	Total
Addition	dd_d_add	10	0	0	10
	dd_dd_add	11	0	0	11
Subtraction	dd_d_sub, d_dd_sub	10	0	0	10
	dd_dd_sub	11	0	0	11
Multiplication	dd_d_mul	14	8	0	22
	dd_dd_mul	15	9	0	24
Division	dd_d_div	16	7	2	25
	d_dd_div	16	8	2	26
	dd_dd_div	17	8	2	27

Algorithm A.21 qd_dd_add $(a_0, a_1, a_2, a_3, b_0, b_1)$

$[c_0, e_0] \leftarrow$ Two-Sum(a_0, b_0)
$[c_1, e_1] \leftarrow$ Two-Sum(a_1, b_1)
$[c_1, e_0] \leftarrow$ Two-Sum(c_1, e_0)
$[c_2, e_1] \leftarrow$ Two-Sum(a_2, e_1)
$[c_2, e_0] \leftarrow$ Two-Sum(c_2, e_0)
$[e_0, e_1] \leftarrow$ Two-Sum(e_0, e_1)
$[c_3, e_0] \leftarrow$ Two-Sum(a_3, e_0)
$e_0 = e_0 \oplus e_1$
$[c_0, c_1, c_2, c_3] \leftarrow$ Renormalize$(c_0, c_1, c_2, c_3, e_0)$
return (c_0, c_1, c_2, c_3)

Algorithm A.22, qd_qd_add, shows the procedure for adding a quad-double number $b_{(qd)}$ to a quad-double number $a_{(qd)}$ and returns the quad-double number $c_{(qd)} = c_0 + c_1 + c_2 + c_3$.

Algorithm A.22 qd_qd_add $(a_0, a_1, a_2, a_3, b_0, b_1, b_2, b_3)$

$[c_0, e_0] \leftarrow$ Two-Sum(a_0, b_0)
$[c_1, e_1] \leftarrow$ Two-Sum(a_1, b_1)
$[c_2, e_2] \leftarrow$ Two-Sum(a_2, b_2)
$[c_3, e_3] \leftarrow$ Two-Sum(a_3, b_3)
$[c_1, e_0] \leftarrow$ Two-Sum(c_1, e_0)
$[c_2, e_0, e_1] \leftarrow$ Three-Sum(c_2, e_1, e_0)
$[c_3, e_0] \leftarrow$ Three-Sum2(c_3, e_2, e_0)
$e_0 = e_0 \oplus e_1 \oplus e_3$
$[c_0, c_1, c_2, c_3] \leftarrow$ Renormalize$(c_0, c_1, c_2, c_3, e_0)$
return (c_0, c_1, c_2, c_3)

A.3.2 subtraction

Algorithm A.23 (A.24), qd_d_sub (d_qd_sub), shows the procedure for subtracting a double-precision number b (a quad-double number $b_{(qd)}$) from a quad-double number $a_{(qd)}$ (a double-precision number a) and returns the quad-double number $c_{(qd)} = c_0 + c_1 + c_2 + c_3$.

Algorithm A.23 qd_d_sub (a_0, a_1, a_2, a_3, b)

$s \leftarrow -b$
$[c_0, c_1, c_2, c_3] \leftarrow$ qd_d_add (a_0, a_1, a_2, a_3, s)
return (c_0, c_1, c_2, c_3)

Algorithm A.24 d_qd_sub (a, b_0, b_1, b_2, b_3)

$s_0 \leftarrow -b_0$
$s_1 \leftarrow -b_1$
$s_2 \leftarrow -b_2$
$s_3 \leftarrow -b_3$
$[c_0, c_1, c_2, c_3] \leftarrow$ d_qd_add (a, s_0, s_1, s_2, s_3)
return (c_0, c_1, c_2, c_3)

Algorithm A.25 (A.26), qd_dd_sub (dd_qd_sub), shows the procedure for subtracting a double-double number $b_{(dd)}$ (a quad-double number $b_{(qd)}$) from a quad-double number $a_{(qd)}$ (a double-double number $a_{(dd)}$) and returns the quad-double number $c_{(qd)} = c_0 + c_1 + c_2 + c_3$.

Algorithm A.25 qd_dd_sub $(a_0, a_1, a_2, a_3, b_0, b_1)$

$b_0 \leftarrow -b_0$
$b_1 \leftarrow -b_1$
$[c_0, c_1, c_2, c_3] \leftarrow$ qd_dd_add $(a_0, a_1, a_2, a_3, b_0, b_1)$
return (c_0, c_1, c_2, c_3)

Algorithm A.26 dd_qd_sub $(a_0, a_1, b_0, b_1, b_2, b_3)$

$b_0 \leftarrow -b_0$
$b_1 \leftarrow -b_1$
$b_2 \leftarrow -b_2$
$b_3 \leftarrow -b_3$
$[c_0, c_1, c_2, c_3] \leftarrow$ dd_qd_add $(a_0, a_1, b_0, b_1, b_2, b_3)$
return (c_0, c_1, c_2, c_3)

Algorithm A.27 shows the procedure for subtracting a quad-double number $b_{(qd)}$ from a quad-double number $a_{(qd)}$ and returns the quad-double number $c_{(qd)} = c_0 + c_1 + c_2 + c_3$.

Algorithm A.27 qd_qd_sub $(a_0, a_1, a_2, a_3, b_0, b_1, b_2, b_3)$

$b_0 \leftarrow -b_0$
$b_1 \leftarrow -b_1$
$b_2 \leftarrow -b_2$
$b_3 \leftarrow -b_3$
$[c_0, c_1, c_2, c_3] \leftarrow$ qd_qd_add $(a_0, a_1, a_2, a_3, b_0, b_1, b_2, b_3)$
return (c_0, c_1, c_2, c_3)

A.3.3 multiplication

Algorithm A.28, qd_d_mul, shows the procedure for multiplying a quad-double number $a_{(qd)}$ by a double-precision number b and returns the quad-double number $c_{(qd)} = c_0 + c_1 + c_2 + c_3$.

Algorithm A.28 qd_d_mul (a_0, a_1, a_2, a_3, b)

$[p_0, q_0] \leftarrow$ Two-Prod(a_0, b)
$[p_1, q_1] \leftarrow$ Two-Prod(a_1, b)
$[p_2, q_2] \leftarrow$ Two-Prod(a_2, b)
$[p_1, q_0] \leftarrow$ Two-Sum(p_1, q_0)
$[p_2, q_0, q_1] \leftarrow$ Three-Sum(p_2, q_0, q_1)
$p_3 \leftarrow a_3 \otimes b$
$[p_3, q_0] \leftarrow$ Three-Sum2(p_3, q_0, q_2)
$q_0 \leftarrow q_0 \oplus q_1$
$[c_0, c_1, c_2, c_3] \leftarrow$ Renormalize$(p_0, p_1, p_2, p_3, q_0)$
return (c_0, c_1, c_2, c_3)

Algorithm A.29, qd_dd_mul, shows the procedure for multiplying a quad-double number $a_{(qd)}$ by a double-double number $b_{(dd)}$ and returns the quad-double number $c_{(qd)} = c_0 + c_1 + c_2 + c_3$.

Algorithm A.29 qd_dd_mul $(a_0, a_1, a_2, a_3, b_0, b_1)$

$[p_0, q_0] \leftarrow$ Two-Prod(a_0, b_0)
$[p_1, q_1] \leftarrow$ Two-Prod(a_0, b_1)
$[p_2, q_2] \leftarrow$ Two-Prod(a_1, b_0)
$[p_3, q_3] \leftarrow$ Two-Prod(a_2, b_0)
$[p_4, q_4] \leftarrow$ Two-Prod(a_1, b_1)
$[p_1, p_2, q_0] \leftarrow$ Three-Sum(p_1, p_2, q_0)
$[p_2, p_3, p_4] \leftarrow$ Three-Sum(p_2, p_3, p_4)
$[q_1, q_2] \leftarrow$ Two-Sum(q_1, q_2)
$[p_2, q_1] \leftarrow$ Two-Sum(p_2, q_1)
$[p_3, q_2] \leftarrow$ Two-Sum(p_3, q_2)
$[p_3, q_1] \leftarrow$ Two-Sum(p_3, q_1)
$p_4 \leftarrow p_4 \otimes q_2 \otimes q_1$
$q_3 \leftarrow q_3 \oplus q_4 \oplus (a_3 \otimes b_0) \oplus (a_2 \otimes b_1)$
$[p_3, q_0] \leftarrow$ Three-Sum2(p_3, q_0, q_3)
$p_4 \leftarrow p_4 \oplus q_0$
$[c_0, c_1, c_2, c_3] \leftarrow$ Renormalize$(p_0, p_1, p_2, p_3, p_4)$
return (c_0, c_1, c_2, c_3)

Algorithm A.30, qd_qd_mul, shows the procedure for multiplying a quad-double number $a_{(qd)}$ by a quad-double number $b_{(qd)}$ and returns the quad-double number $c_{(qd)} = c_0 + c_1 + c_2 + c_3$.

Algorithm A.30 qd_qd_mul$(a_0, a_1, a_2, a_3, b_0, b_1, b_2, b_3)$

$[p_0, q_0] \leftarrow$ Two-Prod(a_0, b_0)
$[p_1, q_1] \leftarrow$ Two-Prod(a_0, b_1)
$[p_2, q_2] \leftarrow$ Two-Prod(a_1, b_0)
$[p_1, p_2, q_0] \leftarrow$ Three-Sum(p_1, p_2, q_0)
$[p_3, q_3] \leftarrow$ Two-Prod(a_0, b_2)
$[p_4, q_4] \leftarrow$ Two-Prod(a_1, b_1)
$[p_5, q_5] \leftarrow$ Two-Prod(a_2, b_0)
$[p_2, q_1, q_2] \leftarrow$ Three-Sum(p_2, q_1, q_2)
$[p_3, p_4, p_5] \leftarrow$ Three-Sum(p_3, p_4, p_5)
$[p_2, p_3] \leftarrow$ Two-Sum(p_2, p_3)
$[p_4, q_1] \leftarrow$ Two-Sum(p_4, q_1)
$[p_3, p_4] \leftarrow$ Two-Sum(p_3, p_4)
$p_4 \leftarrow q_2 \oplus p_5 \oplus q_1 \oplus p_4$
$p_3 \leftarrow p_3 \oplus (a_0 \otimes b_3) \oplus (a_1 \otimes b_2) \oplus (a_2 \otimes b_1) \oplus (a_3 \otimes b_0) \oplus q_0 \oplus q_3 \oplus q_4 \oplus q_5$
$[c_0, c_1, c_2, c_3] \leftarrow$ Renormalize$(p_0, p_1, p_2, p_3, p_4)$
return (c_0, c_1, c_2, c_3)

A.3.4 division

Supposing that $b \neq 0$ and $b_0 \neq 0$. Algorithm A.31 (A.32), qd_d_div (d_qd_div), shows the procedure for dividing a quad-double number $a_{(qd)}$ (a double-precision number a) by a double-precision number b (a quad-double number $b_{(qd)}$) and returns the quad-double number $c_{(qd)} = c_0 + c_1 + c_2 + c_3$.

Algorithm A.31 qd_d_div (a_0, a_1, a_2, a_3, b)

$c_0 \leftarrow a_0 \oslash b$
$[t_0, t_1] \leftarrow$ Two-Prod(c_0, b)
$[r_0, r_1, r_2, r_3] \leftarrow$ qd_dd_sub $(a_0, a_1, a_2, a_3, t_0, t_1)$
$c_1 \leftarrow r_0 \oslash b$
$[t_0, t_1] \leftarrow$ Two-Prod(c_1, b)
$[r_0, r_1, r_2, r_3] \leftarrow$ qd_dd_sub $(r_0, r_1, r_2, r_3, t_0, t_1)$
$c_2 \leftarrow r_0 \oslash b$
$[t_0, t_1] \leftarrow$ Two-Prod(c_2, b)
$[r_0, r_1, r_2, r_3] \leftarrow$ qd_dd_sub $(r_0, r_1, r_2, r_3, t_0, t_1)$
$c_3 \leftarrow r_0 \oslash b$
$[c_0, c_1, c_2, c_3] \leftarrow$ Renormalize2(c_0, c_1, c_2, c_3)
return (c_0, c_1, c_2, c_3)

Algorithm A.32 d_qd_div (a, b_0, b_1, b_2, b_3)

$c_0 \leftarrow a \oslash b_0$
$[t_0, t_1, t_2, t_3] \leftarrow$ d_qd_mul $(c_0, b_0, b_1, b_2, b_3)$
$[r_0, r_1, r_2, r_3] \leftarrow$ d_qd_sub (a, t_0, t_1, t_2, t_3)
$c_1 \leftarrow r_0 \oslash b_0$
$[t_0, t_1, t_2, t_3] \leftarrow$ d_qd_mul $(c_1, b_0, b_1, b_2, b_3)$
$[r_0, r_1, r_2, r_3] \leftarrow$ qd_qd_sub $(r_0, r_1, r_2, r_3, t_0, t_1, t_2, t_3)$
$c_2 \leftarrow r_0 \oslash b_0$
$[t_0, t_1, t_2, t_3] \leftarrow$ d_qd_mul $(c_2, b_0, b_1, b_2, b_3)$
$[r_0, r_1, r_2, r_3] \leftarrow$ qd_qd_sub $(r_0, r_1, r_2, r_3, t_0, t_1, t_2, t_3)$
$c_3 \leftarrow r_0 \oslash b_0$
$[c_0, c_1, c_2, c_3] \leftarrow$ Renormalize2(c_0, c_1, c_2, c_3)
return (c_0, c_1, c_2, c_3)

Supposing that $b_0 \neq 0$. Algorithm A.33 (A.34), qd_dd_div (dd_qd_div), shows the procedure for dividing a quad-double number $a_{(qd)}$ (a double-double number $a_{(dd)}$) by a double-double number $b_{(dd)}$ (a quad-double number $b_{(qd)}$) and returns the quad-double number $c_{(qd)} = c_0 + c_1 + c_2 + c_3$.

Algorithm A.33 qd_dd_div $(a_0, a_1, a_2, a_3, b_0, b_1)$

$c_0 \leftarrow a_0 \oslash b_0$
$[t_0, t_1] \leftarrow$ d_dd_mul (c_0, b_0, b_1)
$[r_0, r_1, r_2, r_3] \leftarrow$ qd_dd_sub $(a_0, a_1, a_2, a_3, t_0, t_1)$
$c_1 \leftarrow r_0 \oslash b$
$[t_0, t_1] \leftarrow$ d_dd_mul (c_1, b_0, b_1)
$[r_0, r_1, r_2, r_3] \leftarrow$ qd_dd_sub $(r_0, r_1, r_2, r_3, t_0, t_1)$
$c_2 \leftarrow r_0 \oslash b$
$[t_0, t_1] \leftarrow$ d_dd_mul (c_2, b_0, b_1)
$[r_0, r_1, r_2, r_3] \leftarrow$ qd_dd_sub $(r_0, r_1, r_2, r_3, t_0, t_1)$
$c_3 \leftarrow r_0 \oslash b$
$[c_0, c_1, c_2, c_3] \leftarrow$ Renormalize2(c_0, c_1, c_2, c_3)
return (c_0, c_1, c_2, c_3)

Algorithm A.34 dd_qd_div $(a_0, a_1, b_0, b_1, b_2, b_3)$

$c_0 \leftarrow a_0 \oslash b_0$
$[t_0, t_1, t_2, t_3] \leftarrow$ d_qd_mul $(c_0, b_0, b_1, b_2, b_3)$
$[r_0, r_1, r_2, r_3] \leftarrow$ dd_qd_sub $(a_0, a_1, t_0, t_1, t_2, t_3)$
$c_1 \leftarrow r_0 \oslash b_0$
$[t_0, t_1, t_2, t_3] \leftarrow$ d_qd_mul $(c_1, b_0, b_1, b_2, b_3)$
$[r_0, r_1, r_2, r_3] \leftarrow$ qd_qd_sub $(r_0, r_1, r_2, r_3, t_0, t_1, t_2, t_3)$
$c_2 \leftarrow r_0 \oslash b_0$
$[t_0, t_1, t_2, t_3] \leftarrow$ d_qd_mul $(c_2, b_0, b_1, b_2, b_3)$
$[r_0, r_1, r_2, r_3] \leftarrow$ qd_qd_sub $(r_0, r_1, r_2, r_3, t_0, t_1, t_2, t_3)$
$c_3 \leftarrow r_0 \oslash b_0$
$[c_0, c_1, c_2, c_3] \leftarrow$ Renormalize2(c_0, c_1, c_2, c_3)
return (c_0, c_1, c_2, c_3)

Supposing that $b_0 \neq 0$. Algorithm A.35, qd_qd_div, shows the procedure for dividing a quad-double number $a_{(qd)}$ by a quad-double number $b_{(qd)}$ and returns the quad-double number $c_{(qd)} = c_0 + c_1 + c_2 + c_3$.

Table 13 shows the number of double precision

Algorithm A.35 qd_qd_div $(a_0, a_1, a_2, a_3, b_0, b_1, b_2, b_3)$

$c_0 \leftarrow a_0 \oslash b_0$
$[t_0, t_1, t_2, t_3] \leftarrow$ d_qd_mul $(c_0, b_0, b_1, b_2, b_3)$
$[r_0, r_1, r_2, r_3] \leftarrow$ qd_qd_sub $(a_0, a_1, a_2, a_3, t_0, t_1, t_2, t_3)$
$c_1 \leftarrow r_0 \oslash b_0$
$[t_0, t_1, t_2, t_3] \leftarrow$ d_qd_mul $(c_1, b_0, b_1, b_2, b_3)$
$[r_0, r_1, r_2, r_3] \leftarrow$ qd_qd_sub $(r_0, r_1, r_2, r_3, t_0, t_1, t_2, t_3)$
$c_2 \leftarrow r_0 \oslash b_0$
$[t_0, t_1, t_2, t_3] \leftarrow$ d_qd_mul $(c_2, b_0, b_1, b_2, b_3)$
$[r_0, r_1, r_2, r_3] \leftarrow$ qd_qd_sub $(r_0, r_1, r_2, r_3, t_0, t_1, t_2, t_3)$
$c_3 \leftarrow r_0 \oslash b_0$
$[c_0, c_1, c_2, c_3] \leftarrow$ Renormalize2(c_0, c_1, c_2, c_3)
return (c_0, c_1, c_2, c_3)

arithmetic operations for quad-double arithmetic. One quad-double arithmetic operation needs tens or hundreds of double-precision operations, then the computation time may require hundreds times greater than double-precision arithmetic.

A.4 The other algorithms for QD

Algorithm A.36, qd_sqr, shows the procedure for squaring a quad-double number $a_{(qd)}$ and returns the quad-double number $c_{(qd)} = c_0 + c_1 + c_2 + c_3$.

Algorithm A.36 qd_sqr (a_0, a_1, a_2, a_3)

$[p_0, q_0] \leftarrow$ Two-Sqr(a_0)
$[p_1, q_1] \leftarrow$ Two-Prod(a_0, a_1)
$p_1 \leftarrow p_1 \otimes 2.0$
$q_1 \leftarrow q_1 \otimes 2.0$
$[p_2, q_2] \leftarrow$ Two-Prod(a_0, a_2)
$p_2 \leftarrow p_2 \otimes 2.0$
$q_2 \leftarrow q_2 \otimes 2.0$
$[p_3, q_3] \leftarrow$ Two-Sqr(a_1)
$[p_1, q_0] \leftarrow$ Two-sum(p_1, q_0)
$[q_0, q_1] \leftarrow$ Two-sum(q_0, q_1)
$[p_2, p_3] \leftarrow$ Two-sum(p_2, p_3)
$[s_0, t_0] \leftarrow$ Two-Sum(q_0, p_2)
$[s_1, t_1] \leftarrow$ Two-Sum(q_1, p_3)
$[s_1, t_0] \leftarrow$ Two-Sum(s_1, t_0)
$t_0 \leftarrow t_0 \oplus t_1$
$[s_1, t_0] \leftarrow$ Fast-Two-Sum(s_1, t_0)
$[p_2, t_0] \leftarrow$ Fast-Two-Sum(s_0, s_1)
$[p_3, q_0] \leftarrow$ Fast-Two-Sum(t_1, t_0)
$p_4 \leftarrow 2.0 \otimes a_0 \otimes a_3$
$p_5 \leftarrow 2.0 \otimes a_1 \otimes a_2$
$[p_4, p_5] \leftarrow$ Two-Sum(p_4, p_5)
$[q_2, q_3] \leftarrow$ Two-Sum(q_2, q_3)
$[t_0, t_1] \leftarrow$ Two-Sum(p_4, q_2)
$t_1 \leftarrow t_1 \oplus p_5 \oplus q_3$
$[p_3, p_4] \leftarrow$ Two-Sum(p_3, t_0)
$p_4 \leftarrow p_4 \oplus q_0 \oplus t_1$
$[c_0, c_1, c_2, c_3] \leftarrow$ Renormalize$(p_0, p_1, p_2, p_3, p_4)$
return (c_0, c_1, c_2, c_3)

Table 13: Number of double precision arithmetic operations for quad-double arithmetic

	Algorithm	$\oplus, \ominus$	$\otimes$	$\oslash$	Total
Addition	qd_d_add	52	0	0	52
	qd_dd_add	71	0	0	71
	qd_qd_add	91	0	0	91
Subtraction	qd_d_sub, d_qd_sub	52	0	0	52
	qd_dd_sub, dd_qd_sub	71	0	0	71
	qd_qd_sub	91	0	0	91
Multiplication	qd_d_mul	96	22	0	118
	qd_dd_mul	154	35	0	189
	qd_qd_mul	171	46	0	217
Division	qd_d_div	261	21	4	286
	d_qd_div	540	66	4	610
	qd_dd_div	273	24	4	301
	dd_qd_div	569	66	4	639
	qd_qd_div	579	66	4	649

Supposing that n is a power of 2. Algorithm A.37, mul_pwr_dd, shows the procedure for multiplying each component of double-double number by n.

Algorithm A.37 mul_pwr_dd (a_0, a_1, n)

$b_0 \leftarrow a_0 \otimes n$
$b_1 \leftarrow a_1 \otimes n$
return (b_0, b_1)

Supposing that n is a power of 2. Algorithm A.38, mul_pwr_qd, shows the procedure for multiplying each component of quad-double number by n.

Algorithm A.38 mul_pwr_qd (a_0, a_1, a_2, a_3, n)

$b_0 \leftarrow a_0 \otimes n$
$b_1 \leftarrow a_1 \otimes n$
$b_2 \leftarrow a_2 \otimes n$
$b_3 \leftarrow a_3 \otimes n$
return (b_0, b_1, b_2, b_3)

Supposing that n is an integer. Algorithm A.39 shows the procedure for computing an n-th power of a quad-double number $a_{(qd)}$ and returns the quad-double number.

Algorithm A.39 qd_pow (a_0, a_1, a_2, a_3, n) : Assume that $n \geq 0$

if $n = 0$ **then**
    return 1.0
end if
if $n = 1$ **then**
    return (a_0, a_1, a_2, a_3)
end if
$r_0 \leftarrow a_0$
$r_1 \leftarrow a_1$
$r_2 \leftarrow a_2$
$r_3 \leftarrow a_3$
$s_0 \leftarrow 1.0$
$N \leftarrow n$
while $N > 0$ **do**
    if N is odd **then**
        $[s_0, s_1, s_2, s_3]$
        $\leftarrow$ qd_qd_mul$(s_0, s_1, s_2, s_3, r_0, r_1, r_2, r_3)$
    end if
    $N \leftarrow N \oslash 2$
    if $N > 0$ **then**
        $[r_0, r_1, r_2, r_3] \leftarrow$ qd_sqr (r_0, r_1, r_2, r_3)
    end if
end while
return (r_0, r_1, r_2, r_3)

References (**Not full**):

[1] T. J. Dekker, A Floating-point technique for extending the available precision, Numerische Mathematik, Vol. 18, pp. 224-242 (1971) .

[2] D. H. Bailey, QD, MPFUN, **http://crd.lbl.gov/˜dhbailey/mpdist/**

[3] Y. Hida, X. S. Li and D. H. Bailey, Quad-double arithmetic: algorithms, implementation, and application, Technical Report LBNL-46996, Lawrence Berkeley National Laboratory, Berkeley, CA 94720 (2000).

[4] Y. Hida, X. S. Li and D. H. Bailey, Algorithms for quad-double precision floating point arithmetic, Proc. of the 15th IEEE Symposium on Computer Arithmetic, pp. 155-162 (2001).

[5] R. Brent, A fortran multiple-precision arithmetic package, ACM TOMS, Vol. 4, No. 1, pp. 71-81 (1978).

[6] J. D. Hogg and J. A. Scott, A fast robust mixed precision solver for the solution of sparse symmetric linear systems, Technical Report, RAL-TR-2008-023 (2008).

[7] T. Saito, E. Ishiwata, and H. Hasegawa, Development of quadruple precision arithmetic toolbox QuPAT on Scilab, Proceedings of the 2010 International Conference on Computational Science and its Applications (ICCSA 2010), Part II, Lecture Notes in Computer Science 6017, pp. 60-70, Springer-Verlag (2010).

[8] T. Saito, E. Ishiwata and H. Hasegawa, Analysis of the GCR method with mixed precision arithmetic using QuPAT, Journal of Computational Science, Volume 3, Issue 3, pp. 87-91, Elsevier (2012).

[9] S. Kikkawa, T. Saito, E. Ishiwata and H. Hasegawa, Development and acceleration of multiple precision arithmetic toolbox MuPAT for Scilab, JSIAM Letters Vol. 5, pp.9-12 (2013).

[10] **MuPAT: http://www.mi.kagu.tus.ac.jp/qupat.html**

[11] Hisashi Kotakemori, Hidehiko Hasegawa, and Akira Nishida, Perfomance Evaluation of Parallel Iterative Method Library using OpenMP, Proc. of the 8th International Conference on High Performance Computing in Asia Pacific Region (HPC Asia 2005), p. 432-436, Nov., 2005, Beijing, China.

[12] Hisashi Kotakemori, Akihiro Fujii, Hidehiko Hasegawa, and Akira Nishida, Implementation of Fast Quad Precision Operation and Acceleration with SSE2 for Iterative Solver Library, IPSJ Transactions on advanced Computing Systems, Vol. 1, No. 1, pp. 73-84 (2008) in Japanese.

[13] Toshiaki Hishinuma, Akihiro Fujii, Teruo Tanaka, and Hidehiko Hasegawa, AVX Acceleration of Sparse Matrix-Vector Multiplication in Double-Double, Proc. on High Performance and Computing Symposium 2013, pp. 23-31 (2013) in Japanese.

[14] R. Barrett, M. Berry, T. F. Chan, J.W. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine and H. Van der Vorst, Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition, SIAM (1994).

[15] **Lis: http://www.ssisc.org/lis/index.en.html**

[16] Tamito Kajiyama, Akira Nukada, Hidehiko Hasegawa, Reiji Suda, and Akira Nishida, SILC: A Flexible and Environment Independent Interface to Matrix Computation Libraries, Lecture Notes in Computer Science 3911, pp. 928-935, Springer-Verlag (2006) (PPAM2005).

[17] **SILC: http://www.ssisc.org/silc/index.html**