AVX2 acceleration of SpMV and vector operations with Double-double precision vectors

Toshiaki Hishinuma, University of Tsukuba
Hidehiko Hasegawa, University of Tsukuba

(OS) High Performance Accurate Computing

- Quality of Computing
- High Performance Computing

+

High Precision Computing

=

High Precision Computing using High Performance Computing

What is DD-AVX Library?

- DD-AVX is Double sparse matrix and DD vector operations tuned for AVX2
  - FMA is also utilized
- DD-AVX has BLAS Level1 (Vector) and Level2 (Matrix-Vector) operations
  - Level1: axpy, axpyz, xpay, dot, nrm2, and scale
  - Level2: SpMV and transposed SpMV (matrix is Double precision)
- DD-AVX has two sparse matrix storage format (CRS / BCRS4x1)
  - BCRS4x1 is good performance on AVX2
- DD-AVX has EasyUI using function and operator overloading in C++
  - Argument of functions does not depend on precision (D/DD)
  - Scalar arithmetics can use C operator (+, -, *, /) by operator overloading
  - Interface of Functions is same for D and DD

DD arithmetic

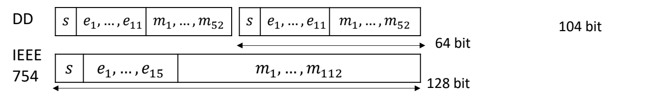
DD number α is represented in combination with 2 double precision numbers α_0 and α_1 as below

$$\alpha = \alpha_0 + \alpha_1$$

- α_0 *is rounded value of* α
- α_1 *is rounded value of* $\alpha - \alpha_0$

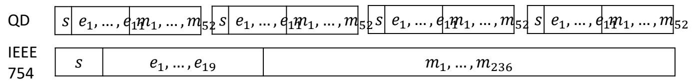
α_0 and α_1 satisfy

$$|\alpha_1| \le \frac{1}{2} ulp(\alpha_0)$$



QD number γ is also represented in the same way *as follows*

$$\gamma = \alpha_0 + \alpha_1 + \alpha_2 + \alpha_3$$

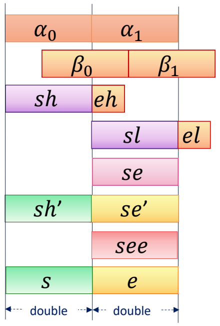


4

Algorithm of DD addition

$\alpha, \beta \in DD, s, e, \alpha_0, \beta_0, \alpha_1, \beta_1 \in Double$

$[s, e] = Addition(\alpha, \beta)$
  $[sh, eh] = twoSum(\alpha_0, \beta_0)$;
  $[sl, el] = twoSum(\alpha_1, \beta_1)$;
  $se = eh + sl$;
  $[sh', se'] = fastTwoSum(sh, se)$;
  $see = se' + el$;
  $[s, e] = fastTwoSum(sh', see)$;
hng



twoSum

```
[s, e] = twoSum(a, b)
    s = a + b;
    v = s - a;
    e = (a - (s - v)) + (b - v);
```

fastTwoSum

```
[s, e] = fastTwoSum(a, b)
    s = a + b;
    e = b - (s - a);
```

5

Algorithm of DD Multiplication

$\alpha, \beta \in DD, p, e, \alpha_0, \beta_0, \alpha_1, \beta_1 \in Double$

$[p, e] = Multiplication(\alpha, \beta)$
  $[p', e'] = twoProd(\alpha_0, \beta_0)$;
  $e' = e + \alpha_0 * \beta_1$;
  $e'' = e' + \alpha_1 * \beta_0$;
  $[p, e] = fastTwoSum(p', e'')$;
hng

twoProd

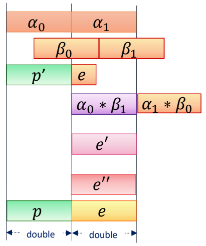
```
[p, e] = twoProd(a, b)
    p = a * b;
    [ah, al] = split(a);
    [bh, bl] = split(b);
    e= (ah * bh - p) + ah * bl
       + al * bh + al * bl;
```

split

```
[h, l] = split(a)
    t = (2^27+1) + a;
    h = t - (t - a);
    l = a - h;
```

fastTwoSum

```
[s, e] = fastTwoSum(a, b)
    s = a + b;
    e = b - (s - a);
```



6

Number of double precision operations

		Add-Subtract	Multiplication	Division	Total
DD	Add-subtract	11	0	0	11
	Multiplication	15	9	0	24
	Division	17	8	2	27
QD	Add-subtract	91	0	0	91
	Multiplication	163	46	0	209
	Division	713	88	5	806

7

Problems

- Heavy:
  The number of double precision operations for DD and QD requires from 10 to 1,000 times
- Difficulty:
  We cannot reduce the number of double precision operations, because the order of computations must be kept!

8

Performance gain of Parallelization

- FMA (Fused Multiply and ADD):
  All (Scalar, Vector, Matrix) , Only for Multiplication,
  not to high, Max 2 times, 1.3 for DD multiplication
- SIMD AVX-2: Vector and Matrix, 4 double precision numbers,
  Max 4 times
- Multithreading OpenMP: Vector and Matrix, for-loop,
  Max **# of cores**

9

DD_ADD_MULT algorithm

University of Tsukuba

DD Addition

```
DD_ADD(a.hi,a.lo,b.hi,b.lo,c.hi,c.lo){
    TWO_SUM(b.hi,c.hi,sh.eh);
    eh = eh + b.lo + c.lo;
    FAST_TWO_SUM(sh,eh,a.hi,a.lo);
}

TWO_SUM(x,y,s,e){
    s = x + y;
    v = s - x;
    e = (x - (s - v)) + (y - v);
}

FAST_TWO_SUM(x,y.s.e){
    s = x + y;
    e = y - (s - x);
}
```

DD Multiplication

```
DD_MULT(a.hi,a.lo,b.hi,b.lo,c){
    TWO_PROD_FMA(b.hi,c,p1,p2);
    p2 += b.lo * c;
    FAST_TWO_SUM(p1,p2,a.hi,a.lo);
}

TWO_PROD_FMA(x,y,p,e){
    p = -x * y;
    e = x * y + p;
    p = -p;
}
```

- DD Add
  - 11 addition and subtraction
- DD Mult
  - 2 FMA
  - 3 multiplication
  - 3 addition and subtraction

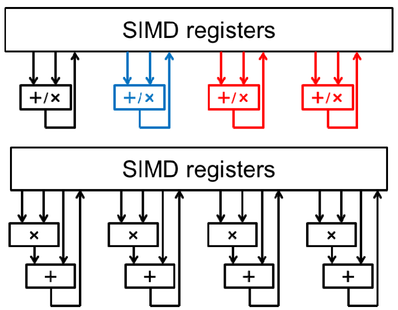
10

SIMD AVX2 (Advanced Vector Extensions 2)

- AVX2 computes 4 FMA instructions in parallel

Scalar
SSE2
AVX

AVX2



11

Programming using DD-AVX Library

There are 5 data types:
Double {Scalar, Vector, Matrix}
DD {Scalar, Vector}

Scalar arithmetics can use C operator

Functions have same interface.

```
#include<dd-avx.hpp>
int main(){
  D_Scalar alpha = 1.0
  DD_Scalar beta=5.0, gamma=0.0;
  D_Vector x;
  DD_Vector y;
  D_Matrix A;
  DD_AVX_input(A, "input.mtx", "bcrs4x1");
  DD_AVX_vector_calloc(x, A.N);
  DD_AVX_vector_calloc(y, A.N);
  gamma = beta * 5.0 + DD(alpha);
  axpy(-gamma, x, y) //y += -gamma * x
  spmv(A,x,y)
  gamma.print();
}
```

12

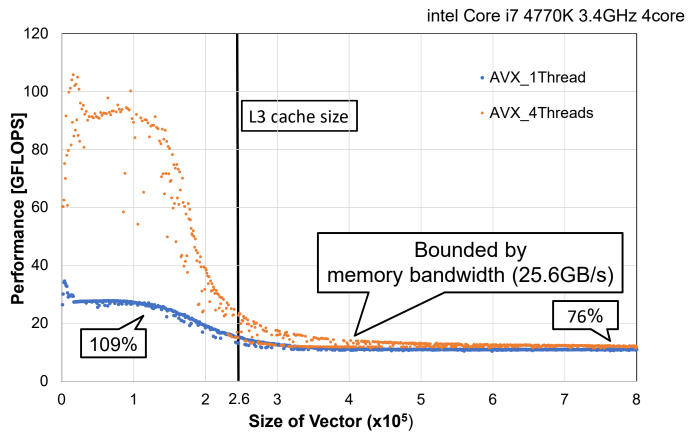
List of Operations

operations
void DD_AVX_axpy(X_Scalar alpha, X_Vector vx, X_Vector vy);
void DD_AVX_axpyz(X_Scalar alpha, X_Vector vx, X_Vector vy, X_Vector vz);
void DD_AVX_dot(X_Vector vx, X_Vector vy, X_Scalar* val);
void DD_AVX_nrm2(X_Vector vx, X_Scalar* val);
void DD_AVX_xpay(X_Vector vx, X_Scalar alpha, X_Vector vy);
void DD_AVX_scale(X_Scalar alpha, X_Vector vx);
void DD_AVX_SpMV(X_Matrix A, X_Vector vx, X_Vector vy);
void DD_AVX_TSpMV(X_Matrix A, X_Vector vx, X_Vector vy);

List of Functions

Scalar type	Vector type	D_Matrix type
void print()	X_Vector operator=(X_Vector vec);	void input(const char *filename);
	X_Vector copy(X_Vector Vec);	
Scalar operator=(T);		void convert_crs2bcrs();
Scalar operator-();	void malloc(int n);	void convert_crs2bcrs();
	void free();	
X_Scalar operator+(T1,T2);		void free();
X_Scalar operator-(T1,T2);	void print(int n);	
X_Scalar operator*(T1,T2);	void print_all();	
X_Scalar operator/(T1,T2);	int getsize();	
X_Scalar dot(X_Vector x, X_Vector y);	void input_plane_format(const char *filename);	
X_Scalar nrm2(X_Vector x);	void input_mm_format(const char *filename);	
	void output_plane_format(const char* file);	
	void output_mm_format(const char* file);	
	void broadcast(T val); // set val to all	

Performance of DD-Vector Operation (axpy)



bytes/flop of $\boldsymbol{y}_{DD} = A_D \boldsymbol{x}_{DD}$

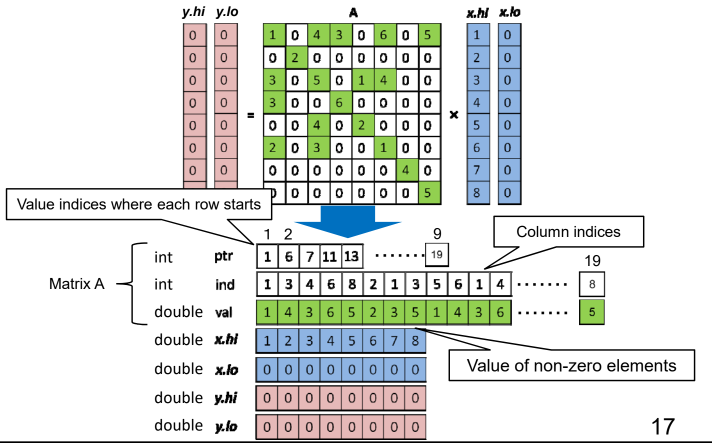
- **In DD-AVX, matrix A is double precision**
  - In many cases, input matrix A will be given by double precision
  - Data size is a half
  - $\boldsymbol{y}_{DD} = A_D \boldsymbol{x}_{DD}$ can reduce the amount of data and bytes/flop

Table 1 bytes/flop of $\boldsymbol{y} = A\boldsymbol{x}$

	bytes / flops
$\boldsymbol{y}_D = A_D \boldsymbol{x}_D$	14 (28 bytes / 2 flops)
$\boldsymbol{y}_{DD} = A_{DD} \boldsymbol{x}_{DD}$	2.26 (52 bytes / 23 flops)
$\boldsymbol{y}_{DD} = A_D \boldsymbol{x}_{DD}$	2.09 (44 bytes / 21 flops)

94%

CRS format (compressed row storage)



DD-SpMV in CRS without SIMD (Scalar)


```
for (i = 1 ; i <= N ; i++)
{
   for (j = ptr[i] ; j < ptr[i+1]; j++)
   {
      y[i] += val[ j ] * x[index[ j ]];
   }
}
```

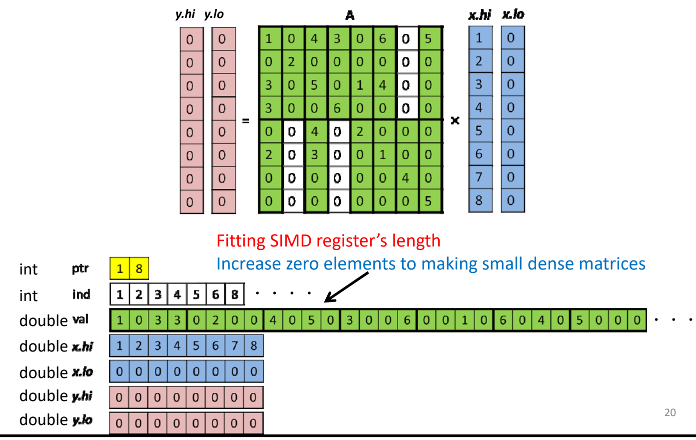
- Row-wise access
- Memory access of x is indirect

DD-SpMV in CRS using AVX2


```
for (i = 1 ; i <= N ; i++)
{
   yv2 = setzero_pd();  //initializing yv
   for (j = ptr[i] ; j < ptr[i+1] - 3 ; j+=4)
   {
      av1 = load(&val[ j ]);
      xv2 = set(x[ind[j]], x[ind[j+1]], x[ind[j+2]], x[ind[j+3]]]);
      DD_ADD_MULT(yv2, av1, xv2)
   }
   av1 = load(&val[ j ]);
   xv2 = FRACTION_PROCESSING();
   DD_ADD_MULT(yv2, av1, xv2)
   y[i] =  REDUCTION(yv2);
}
```

- av_1 is SIMD register type value
- xv_2 and yv_2 are two SIMD registers for DD precision
- x and y are DD precision array

BCRS4x1 format (block size is 4×1)



DD-SpMV in BCRS4x1 using AVX2

University of Tsukuba

CRS

```
for (i = 1 ; i <= N ; i++)
{
   yv2 = setzero_pd();  //initializing yv
   for (j = ptr[i] ; j < ptr[i+1] - 3 ; j+=4)
   {
      av1 = load(&val[ j ]);
      xv2 = set(x[ind[j]], x[ind[j+1]],
      x[ind[j+2]], x[ind[j+3]]]);
      DD_ADD_MULT(yv2, av1, xv2)
   }
   av1 = load(&val[ j ]);
   xv2 = FRACTION_PROCESSING();
   DD_ADD_MULT(yv2, av1, xv2)
   y[i] =  REDUCTION(yv2);
}
```

BCRS4x1

```
for (i = 1 ; i <= N-3; i+=4)
{
   yv2 = setzero_pd(); //initialize yv
   for (j = ptr[i] ; j < ptr[i+1] ; j++)
   {
      av1 = load(&val[ 1+4*(j-1) ]);
      xv2 =broadcast( x[ind[j] ] );

      DD_ADD_MULT(yv2, av1, xv2);
   }
   store(&y[i], yv2);
}
```

BCRS4x1 (row=4, col=1)

- column-wise loop unrolling
- broadcast x
- continuous storing y (store)

BCRS4x1 can eliminate performance degradation factors

21

Bytes/flop in BCRS4x1 using AVX2

University of Tsukuba

In j-loop, bytes/flop of different between CRS and BCRS4x1

CRS:

4×8 (A.val) + 4×16 ($\boldsymbol{x}$) + 4×4(A.ind) = 112 bytes

BCRS4x1:

4×8 (A.val) + 1×16 ($\boldsymbol{x}$) + 4×1(A.ind) = 52 bytes

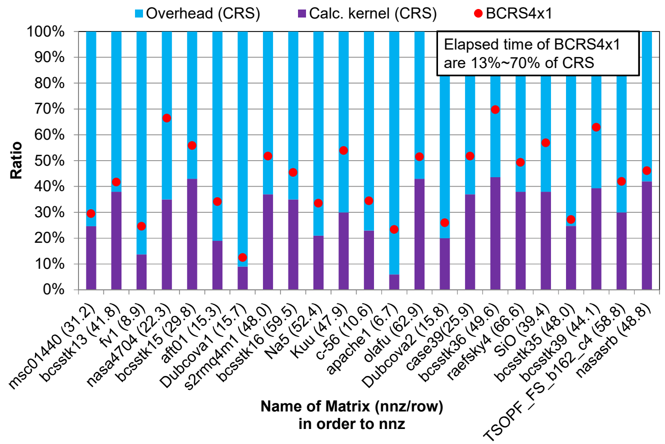
Bytes / flop of $\boldsymbol{y}_{DD} = A_D \boldsymbol{x}_{DD}$

	Byte / flop
Scalar CRS	1.47 (28 bytes / 19 flops)
AVX2 CRS	5.88 (112 bytes / 19 flops)
◎ AVX2 BCRS4x1	2.73 (52bytes / 19 flops)

22

Improvement by BCRS4x1 for DD-SpMV

University of Tsukuba



23

DD-TSpMV in CRS without SIMD (Scalar)

University of Tsukuba

```
for (i = 1 ; i <= N ; i++)
{
   for (j = ptr[i] ; j < ptr[i+1]; j++)
   {
      y[ i ] += val[ j ] * x[index[ j ]];
   }
}
```

```
for (i = 1 ; i <= N ; i++)
{
   for (j = ptr[i] ; j < ptr[i+1]; j++)
   {
      y[index[ j ]] += val[ j ] * x[ i ];
   }
}
```

- Access pattern of A is same as DD-SpMV
- DD-SpMV
  - Memory access of $\boldsymbol{x}$ is indirect
- DD-TSpMV
  - Memory access of $\boldsymbol{y}$ is indirect

24

DD-TSpMV using AVX2 in BCRS4x1


```
#pragma omp parallel private(work, jb, av, xv, yv){
for (ib = 1 ; ib <= brock_row ; ib++){
   xv2 = load(x[ib]):
   for (jb = ptr[ib] ; jb < ptr[ib+1] -3 ; jb++)
   {
         av1 = load(&val[jb]);
         yv2 = broadcast(work[ind[jb]);
         DD_ADD_MUL(yv2, av1, xv2)
         work_vec[i] = REDUCTION(yv1);
}}}}
summation_of_work_vectors();
```

- REDCUTION for storing y is needed for each column
  - REDUCTION requires 33 flops (DD_ADD_MUL is 19 flops)
  - It is very costly

Multi-threading of DD-TSpMV

- CRS has row-wise access pattern
  - Needs to access all elements of y in j-loop to store
  - Transposed SpMV needs work vector for each thread to store y
- Initialization and Summation of work vectors are needed.

Transposed SpMV in multi-thread

```
initialization(work) // thread_num * 16 bytes
#pragma omp parallel for private (i, j , work)
for(i=1 ; i<=N ; i++)
      for(j=ptr[ i ] ; j<ptr[ i+1 ] ; j++)
            work[ index[ j ] ] = work [ A[ j ] ] + val[ j ] * x[ i ]

for(i=1;i<=N,i++) // summation of work vectors
     y[i] = work[ind[j+0+thread_num*N] + ...
                        + work[ind[jj+3+thread_num * N]];
```


Improvement multi-threading for DD-TSpMV

- Idea: column-wise multi-threading
  - Thread partition for j-loop
- Column-wise multi-threading is difficult for DD-TSpMV in CRS and BCRS1x4
  - Because, AVX2 needs four double precision operation simultaneously
- BCRS4x1 only computes one column in j-loop
  - It can be thread-partitioned easily.

Modified DD-TSpMV in BCRS4x1


```
k= omp_num_threads()
#pragma omp parallel private(work, jb, av, xv, yv){
alpha = N / num_threads * k
beta = N / num_threads * (k+1)
for (ib = 1 ; ib <= brock_row ; ib++){
   xv2 = load(x[ib]):
   #pragma omp for
   for (jb = ptr[ib] ; jb < ptr[ib+1] -3 ; jb++)
   {
      If ( alpha < ind[jb] <= beta)  //thread-partitioning
      {
         av1 = load(&val[jb]);
         yv2 = broadcast(work[ind[jb]);
         DD_ADD_MULT(yv2, av1, av2)
         store(&work[num_threads][ib] , yv2);
      }
   }
}
```

alpha and *beta* are thread separator

load/store is good

・column-wise multi-threading
・BCRS4x1 only computes one column in j-loop
・BCRS4x1 can be thread-partitioned easily.

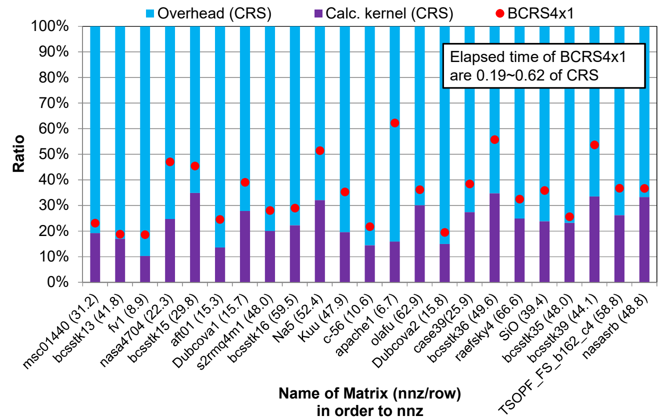
Features of DD-TSpMV

	BCRS1x4	BCRS4x1	
		row wise	**col. wise**
Loading x	broadcast	load	load
Loading y	load	broadcast	broadcast
Storing y	Store	REDUCTION	store
Fraction processing	None	none	none
Computation ratio (max)	x4	x4	x4
work vector	thread num	thread num	none

performance degradation factors

- Column-wise multithreading for BCRS4x1 is good
  - Thread-partitioning is easy
  - Memory access is smooth

Effect of BCRS4x1 for DD-TSpMV (column-wise multi-threading)



Performance of DD-SpMV and DD-TSpMV

Table Elapsed time of SpMV and TSpMV using bandmatrix [ms] (N=10^5,bandwidth=32)

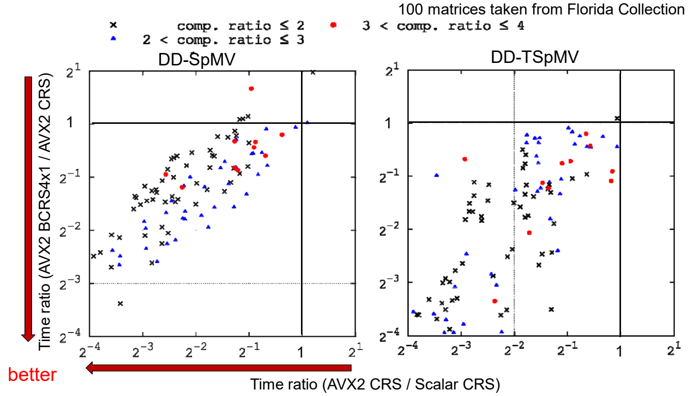
	DD-SpMV	DD-TSpMV
CRS	2.14	3.97
BCRS1x4	2.02	2.94
BCRS4x1 (row-wise)	1.74	13.31
BCRS4x1 (col-wise)	none	2.41

- DD-SpMV : BCRS4x1 is the best
- DD-TSpMV : BCRS4x1 in column-wise is the best
- BCRS4x1 is good

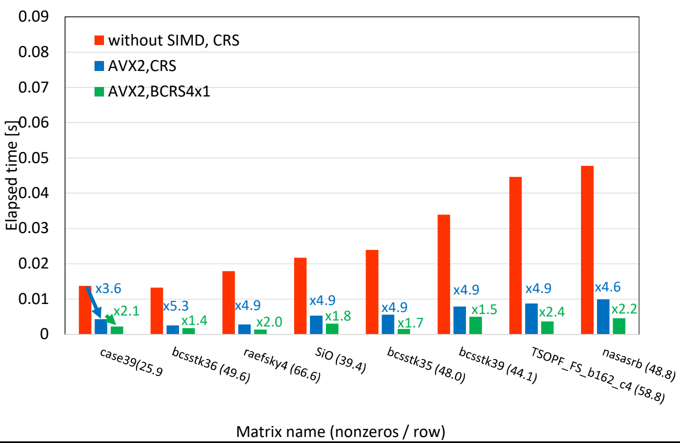
(intel Core i7 4770 4core)

Computation time ratio of DD-SpMV and DD-TSpMV

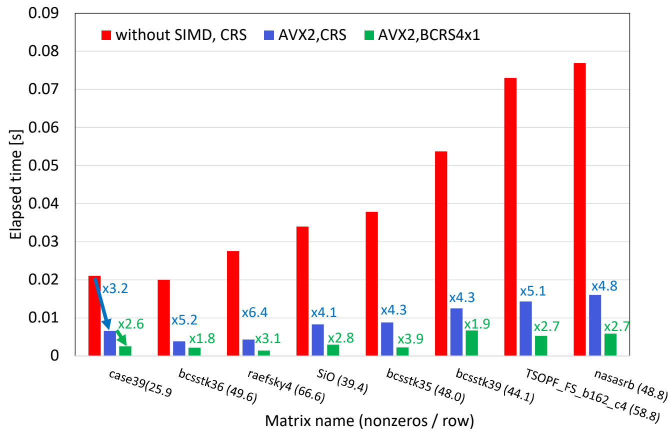
100 matrices taken from Florida Collection



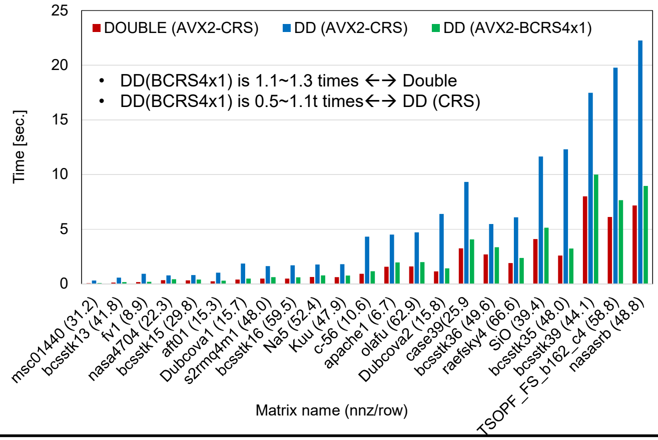
Performances of $y_{DD} = A_D x_{DD}$



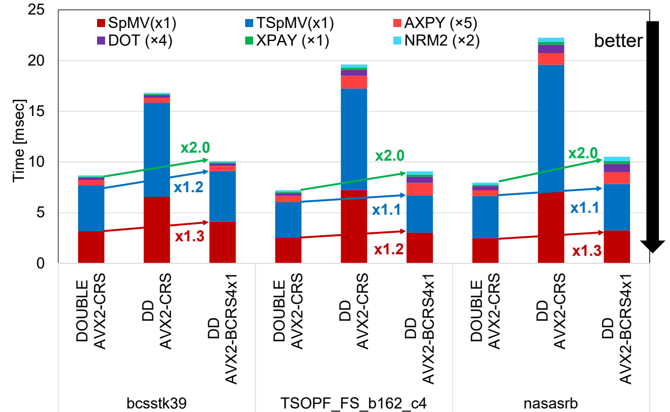
Performances of $y_{DD} = A_D^T x_{DD}$



100 iterations of BiCG method

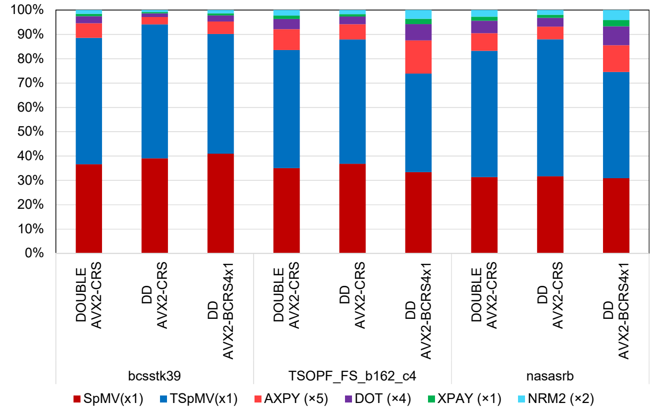


100 iterations of BiCG method



100 iterations of BiCG method

University of Tsukuba



37

DD-AVX Library

- Double sparse matrix and DD vector operations
- FMA, AVX2 and OpenMP
- Double precision for Matrices
  - Small Memory Space
  - Reducing Memory Access
- BCRS4x1 for AVX2
- Good Performance?

Summary

- Computation time of Vector operations are more than 2 times of double because of memory access.
- Bytes/flops is a big problem!
- SpMV and TSpMV with Double-Matrix becomes less than 2 times.
- Sparse Storage format : BCRS4x1 is good
- SpMV 1.1~1.3 times of Double
- TSpMV 1.1~1.2 times of Double
- At most 2 times when data size is 4 times

More Reduction of computation time

- The computation cost of DD-precision SpMV and transposed SpMV on AVX2 is 3 times of that of double.
- There is no room to speed up DD operations.
- Use more Double operations instead of DD operations.
- HYBRID - Mixed Precision Iterative Methods

Strategy of Mixed-precision

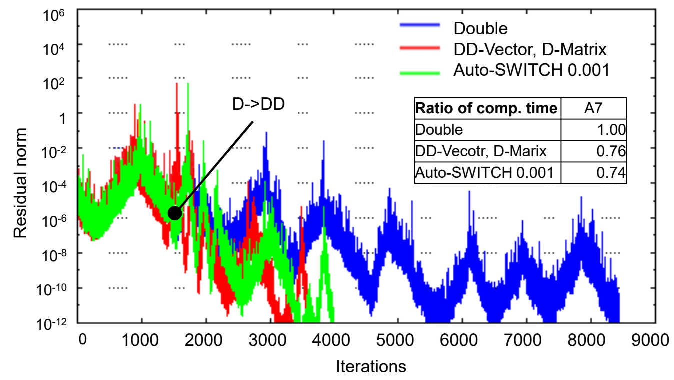
- Combination of Double and DD precisions in each iteration step
- DQ-SWITCH
  - Current solution xk is passed at the restart
  - Only Upper part is used for Double Precision
    (Upper and Lower part are stored in different arrays)
- Automatic restart for DQ-SWITCH
  - Compute deviation of residual norm and restart
  - p : number of samples (a memory of history)
  - v : How large improvement of residual
- Full DD-precision

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

(4 nodes, intel Core i7 4770 4core 3.4GHz (8MB, 16GB), Fedora21, intel C/C++ Compiler 13.0.1)

Case of test matrix A7

University of Tsukuba



Ratio of comp. time	A7
Double	1.00
DD-Vecotr, D-Marix	0.76
Auto-SWITCH 0.001	0.74

BiCG method

University of Tsukuba

1. Set an initial guess $\boldsymbol{x}_0$
2. Compute $\boldsymbol{r}_0 = \boldsymbol{b} - A\boldsymbol{x}_0$
3. Set an arbitrary vector $\boldsymbol{r}_0$ s.t. $(\boldsymbol{r}_0, \boldsymbol{r}^*_0) \neq 0$ e.g., $\boldsymbol{r}^*_0 = \boldsymbol{r}_0$
4. Set $\boldsymbol{p}_0 = \boldsymbol{r}_0$, $\boldsymbol{p}^*_0 = \boldsymbol{r}^*_0$
5. For k = 0,1,2,…
6. $\alpha_k = (\boldsymbol{r}^*_k, \boldsymbol{r}_k) / (\boldsymbol{p}^*_k, A\boldsymbol{p}_k)$
7. $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k$
8. $\boldsymbol{r}_{k+1} = \boldsymbol{r}_k - \alpha_k A\boldsymbol{p}_k$
9. $\boldsymbol{r}^*_{k+1} = \boldsymbol{r}^*_k - \alpha_k A^h\boldsymbol{p}^*_k$
10. $\beta_k = (\boldsymbol{r}^*_{k+1}, \boldsymbol{r}_{k+1}) / (\boldsymbol{r}^*_k, \boldsymbol{r}_k)$
11. $\boldsymbol{p}_{k+1} = \boldsymbol{r}_{k+1} + \beta_k \boldsymbol{p}_k$
12. $\boldsymbol{p}^*_{k+1} = \boldsymbol{r}^*_{k+1} + \beta_k \boldsymbol{p}^*_k$
13. End For

$\boldsymbol{q} = A\boldsymbol{p}_k$
$\boldsymbol{q}^* = A^H\boldsymbol{p}^*_k$

Candidates

University of Tsukuba

- All Double : fast and un-stable
- All DD : stable and not-fast
- DD-Vector, Double-Matrix
- Mix1: A, ***x***, ***b***, ***r***, ***r**** in Double
- Mix2: A, ***x***, ***b***, ***r***, ***r****, ***p**** in Double
- Mix3: ***A***, ***x***, ***b***, ***r***, ***r****, ***p*** in Double
- Automatic SWITCH from D to DD (10^{-1})
- Automatic SWITCH from D to DD (10^{-2})
- SWITCH from D to DD when p=100, v<10^{-1}
- SWITCH from D to DD when p=100, v< 10^{-2}

Result

	ASIC_100ks (N = 99,190)	TSOPF_RS_b39_c7 (N = 141,098)	memplus (N = 17,758)	epb3 (N = 84,617)
All Double	3371(3.2s)	6204(2.5s)	∞	∞
p : DD	3156(3.8s)	4043(1.7s)	∞	∞
p*: DD	3693(4.5s)	5789(2.4s)	12129(5.0s)	∞
p and p*: DD	3240(4.2s)	3871(1.9s)	11613(5.7s)	13528(50.8s)
Vectors : DD	3011(2.7s)	3646(1.8s)	10938(5.4s)	10432(35.9s)
Full DD	3011(5.8s)	3646(4.1s)	10938(12.3s)	10434(78.8s)
DQ-SWITCH	3036(2.8s)	3863(2.0s)	11589(6.1s)	11756(33.2s)

Some special problem for MIX

N=586,358, nnz=43,749,816,　nnz/row=74.6

	Double	DD (CRS)	DD (BCRS)	Mix (BCRS)
Iter.	170,767	73,551	72,663	80,231
Time [s]	3,620	2,431	2,135	2,011

- Mix is fastest.
  - Elapsed time in Mix at 1 iter. is 80% that in DD (BCRS).
  - # of iter. of Mix increases 1.1 times by DD (BCRS)
    But, 5% faster.

Testing bed

- CPU : intel Core i7 4770 4core 3.4GHz
  - L3 cache : 8MB
- Memory : 16GB (8GB × 2 dual channel)
  - Bandwidth : 12.6 [GB/s] × 2 = 25.2 GB/s
- OS : Fedora21
- Compiler : intel C/C++ Compiler 13.0.1
  - Options : *–O3 –xCORE-AVX2 –openmp –fp-model precise*
  - OpenMP scheduling is guided

Conclusion

- Partial use has small improvement; sometimes not converge.
- DD-precision is robust, but costly.
- DD-precision except matrices is robust and reasonable cost.
- DQ-SWITCH may have improvement in keeping robustness.
- Automatic restart is not easy.
  BiCG has no special property to detect its stagnation.
- Mixed precision iterative methods will be practically useful.