Quadruple-precision BLAS using Bailey's arithmetic with FMA instruction : Its performance and applications

iWAPT 2017

June 2, 2017

Susumu YAMADA[1]
Toshiyuki IMAMURA[2], Takuya Ina[1], Yasuhiro IDOMURA[1],
Narimasa SASA[1], Masahiko MACHIDA [1]

[1] Center for Computational Science and e-Systems, Japan Atomic Energy Agency
[2] AICS, RIKEN

Outline

- ✓ Introduction of quadruple precision operations
- ✓ Bailey's double-double arithmetic
- ✓ Speedup for Bailey's double-double arithmetic by FMA instruction
- ✓ Quad-precision Eigenvalue solver using Quad-precision BLAS+FMA
- ✓ Performance tuning for Quad-precision BLAS+FMA
- ✓ Conclusions

Necessity of quad-precision operations

- Nowadays, we can execute ultra-huge-scale simulations on ultra-large parallel computers.
- Since the huge simulations require a great number of the floating-point operations, there would be a possibility that the rounding error accumulates so much until its result has little validity.
- Actually, when we solved the eigenvalue problem for 375 000-dimentional matrix using the direct method with the real(8) values, the result has only a few digit of accuracy[1].

One of the strategies to avoid the precision issue is to use quad-precision values.

[1] S. Yamada, T. Imamura, T. Kano, and M. Machida, High-Performance Computing for Exact Numerical Approaches to Quantum Many-Body Problems on the Earth Simulator, *Proc. of SC06* (2006).

Two quad-precision formats

Real(16) format

implement: very easy
accuracy: about 34 decimal digits
speed: **very slow (software implementation**)

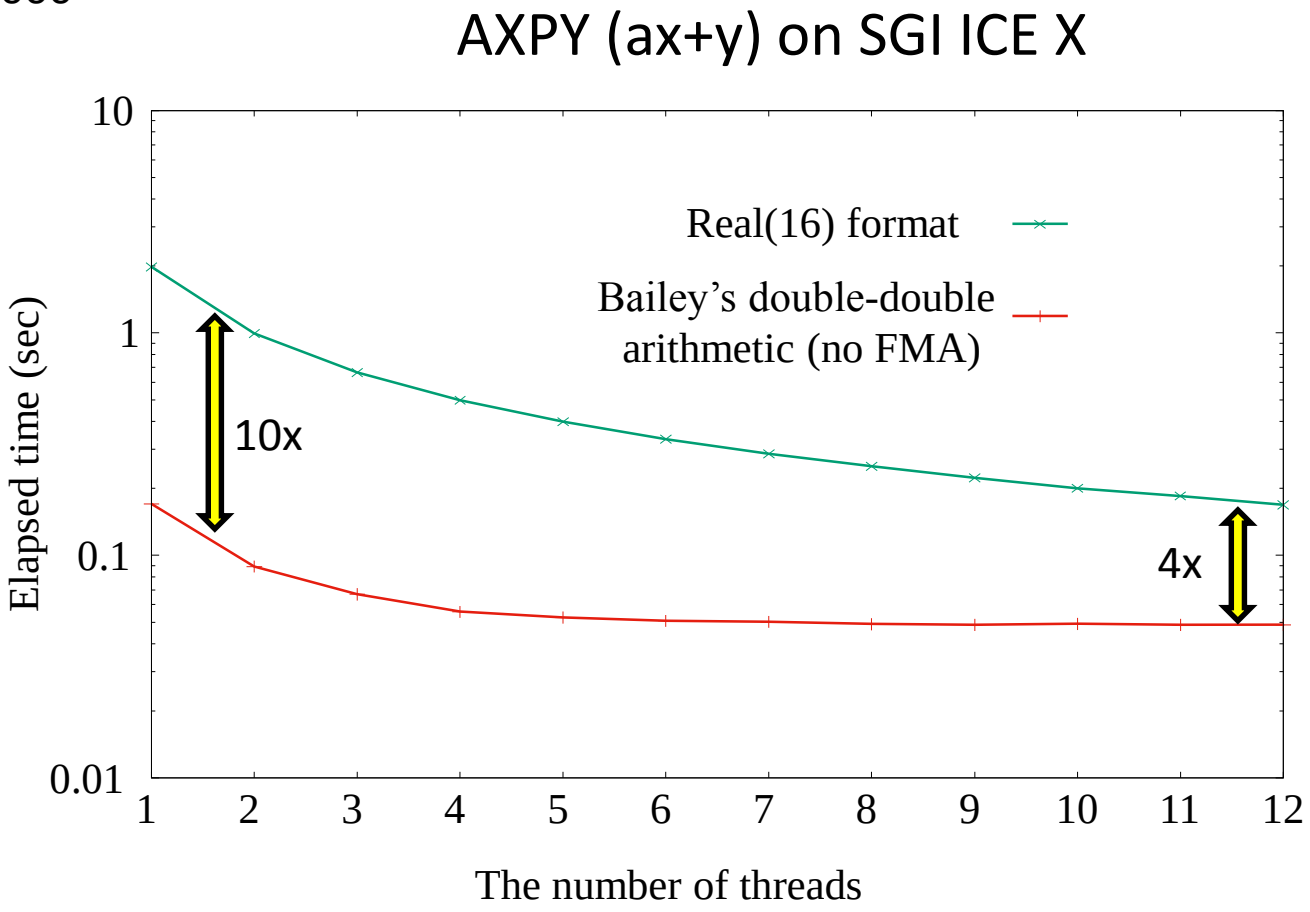
double-double arithmetic (by D.H. Bailey)

implement: easy
accuracy: about 32 decimal digits
speed: **high (hardware implementation)**

A quadrature-precision value is implemented using a pair of double-precision values.

Performance comparison with Real(16) format and Bailey's arithmetic

N=51,200,000



When the number of the threads is small, Bailey's arithmetic is about 10 times faster than real(16) format. When the number is 12, about 4 times faster.

Quadruple Precision BLAS

BLAS（Basic Linear Algebra Subprograms)

40 routines

We developed quad-precision BLAS using Bailey's double-double arithmetic.

QPBLAS（Quadruple Precision Basic Linear Algebra Subprograms)

Quad-precision algorithm using Bailey's arithmetic

Addition C=A+B using Bailey's double-double arithmetic

```
subroutine dd_add(ch,cl,ah,al,bh,bl)
implicit none
double precision ch, cl, ah, al, bh, bl
double precision p1, p2, s1
p1 = ah + al
p2 = p1 - ah
cl = (ah -(p1-p2)) + (bh - p2)
ch = p1
cl = cl +al + bl
s1 = ch + cl
cl =cl - (s1 -ch)
ch =s1
end
```

11 floating-point operations

Multiplication C=A * B using Bailey's double-double arithmetic

```
subroutine dd_mul(ch,cl,ah,al,bh,bl)
implicit none
double precision ch, cl, ah, al, bh, bl
double precision u1, p1, p2, r1, r2, s1
ch = ah *bh
u1 = 134217729.0*ah
p1 = u1 - (u1 - ah)
p2 = ah - p1
u1 = 134217729.0*bh
r1 = u1 - (u1 - bh)
r2 = bh - r1
cl = ((p1*r1-ch) + p1*r2 + p2*r1) + p2*r2
cl = cl + (ah * bl +al * bh)
s1 = ch +cl
cl =cl - (s1 -ch)
ch =s1
end
```

24 floating-point operations

Bailey's arithmetic using FMA instruction

FMA instruction

For double precision values a, b, and x,

a*x+b is calculated using 128-bit values and the result with a 64-bit value is obtained.

When we utilize the FAM instruction, we can realize the Bailey's multiplication with fewer operations.

```
void bailey_dd_mul_fma( x_hi, x_lo, y_hi, y_lo, z_hi, z_lo )
   double x_hi, x_lo, y_hi, y_lo, *z_hi, *z_lo;
{
   double p1, p2;

   p1 = (x_hi) * (y_hi);
   p2 = fma( (x_hi), (y_hi), -p1 );
   p2 += ( (x_lo) * (y_hi) + (x_hi) * (y_lo) );
   *z_hi = p1 +p2;
   *z_lo = p2 - ( (*z_hi) -p1 );

   return;
}
```

10 floating-point operations

Original method requires **24 floating-point operations.**

The multiplication using FMA is realized with only **10 floating-point operations.**

The decrease is **about 60%.**

Reduction of calculation time

Replace the multiplication operations in QPBLAS by the operations using FMA instruction.

Number of instructions of DD arithmetic

When we utilize a compiler optimization, the multiply-accumulation operation is executed by one instruction.

Quad-precision multiplication using Bailey's arithmetic without FMA instruction

```
subroutine dd_mul(ch,cl,ah,al,bh,bl)
implicit none
double precision ch, cl, ah, al, bh, bl
double precision u1, p1, p2, r1, r2, s1
ch = ah * bh
u1 = 134217729.d0*ah
p1 = u1 - (u1 - ah)
p2 = ah - p1
u1 = 134217729.0*bh
r1 = u1 - (u1 - bh)
r2 = bh - r1
cl = ((p1*r1-ch) + p1*r2 + p2*r1) + p2*r2
cl = cl + ah * bl +al * bh
s1 = ch + cl
cl =cl - (s1 -ch)
ch =s1
end
```

6 multiply-accumulation operations

No FMA: 18 instructions

Quad-precision multiplication using Bailey's arithmetic with FMA instruction

```
void dd_mul_fma(ch,cl,ah,al,bh,bl)
double ah,al,bh,bl,*ch,*cl;
{
double p1, p2;
p1 = ah * bh;
p2 = fma(ah,bh,-p1);
p2 += al*bl + ah*bl
*ch = p1 + p2;
*cl = p2 - (*ch - p1);
return;
}
```

3 multiply-accumulation operations

With FMA: 7 instructions

Theoretical elapsed time

$$\text{Theoretical elapsed time (sec)} = \max(cal., comm.)$$

Calculation time

$$cal. = \frac{\#\text{add}*11+\#\text{mul}*(18\,(\text{no FMA})\,\text{or}\,7(\text{FMA}))}{\#\text{Clock}*\#\,\text{INST}*\#\,\text{THREADS}}$$

Time for data transferred from memory

$$comm. = \frac{\text{The number of transferred data (byte)}}{\text{Memory bandwidth (byte/sec)}}$$

#add : the number of quad-precision additions

#mul : the number of quad-precision multiplications

#Clock : the clock rate of the processor

#INST : the number of the executable instructions per cycle per clock

#THREADS : the number of the threads

Details of computers

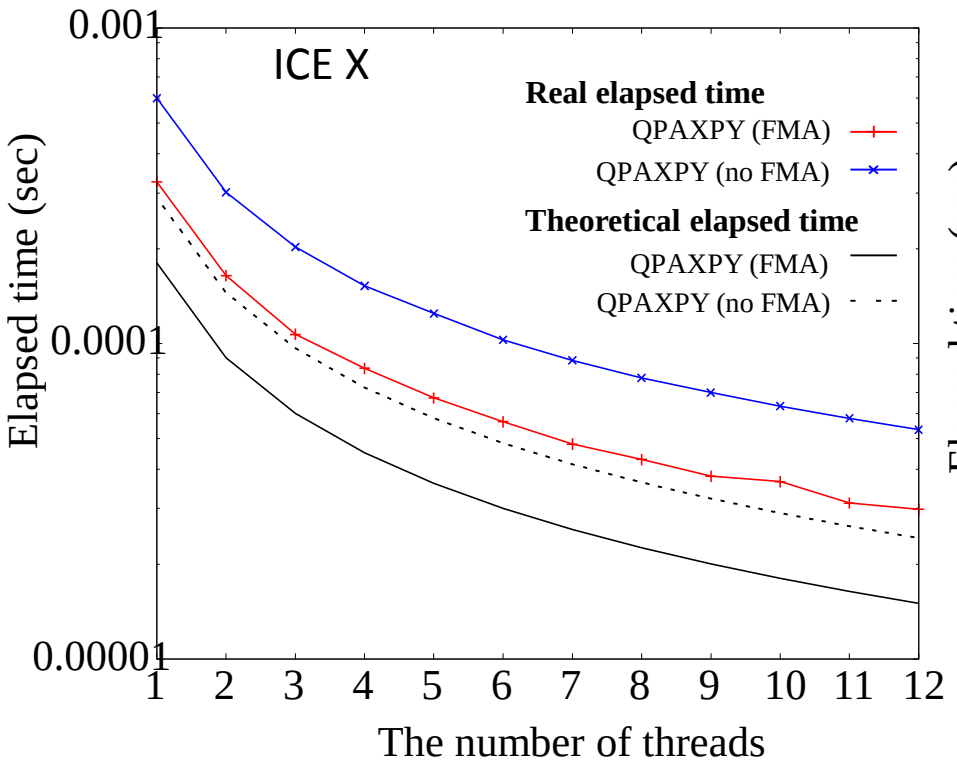
	SGI ICE X (Japan Atomic Energy Agency)	FUJITSU FX10 (The University of Tokyo)
CPU	Intel Xeon E5-2680 v3 (2.5GHz, 30MB cache)	SPARC64™IXfx (1.848GHz, 12MB L2 cache)
Number of executable instructions per cycle per clock (#INST)	8	4
Number of cores per CPU	12	16
Number of CPUs per node	2 CPUs	1 CPU
Memory per node	64GB	32GB
Memory bandwidth	68GB/s	85GB/s
Network	Infini Band	Torus network (Tofu)
Network Bandwidth	6.8GB/s	5GB/s
Complier	Intel compiler	Fujitsu compiler

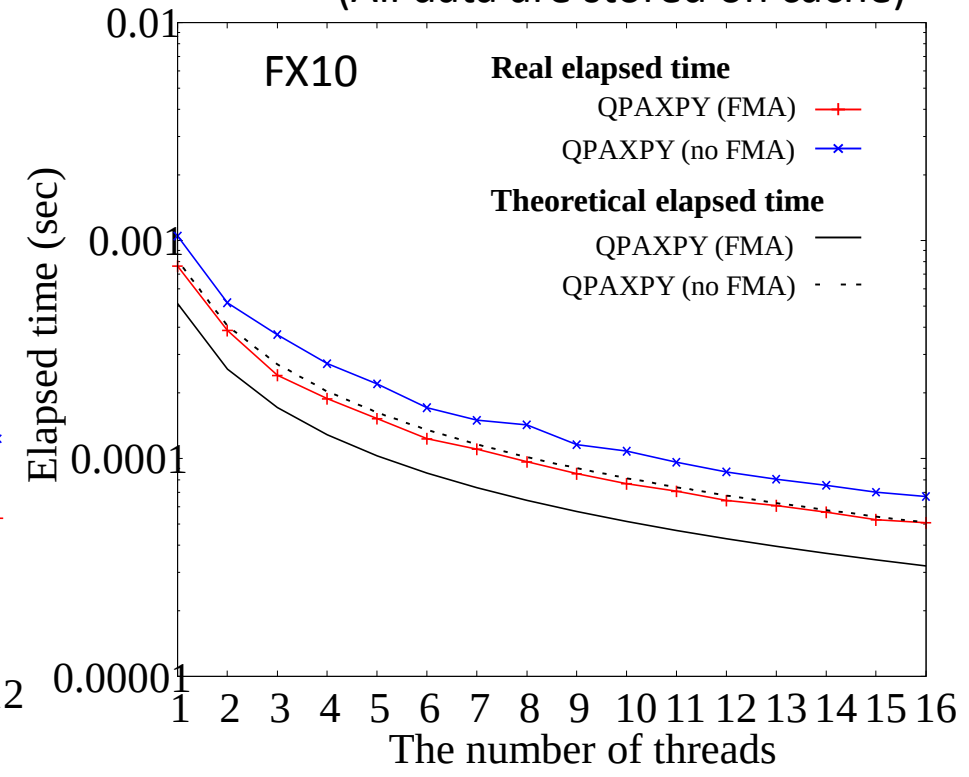
Elapsed time of Level 1 (small size)

AXPY

N=200,000

(All data are stored on cache)



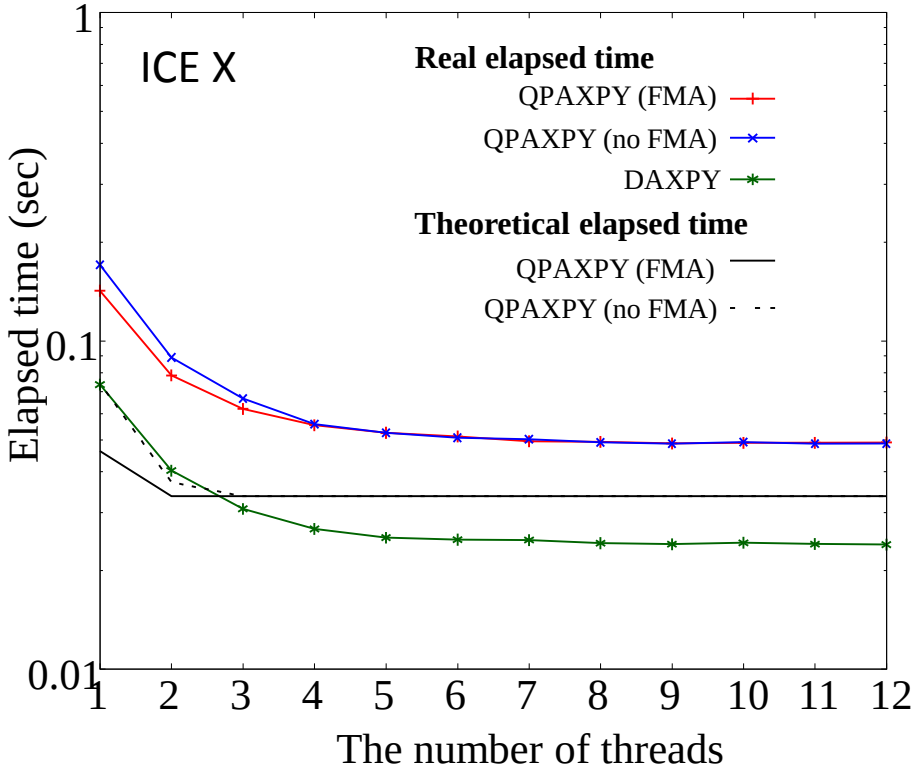


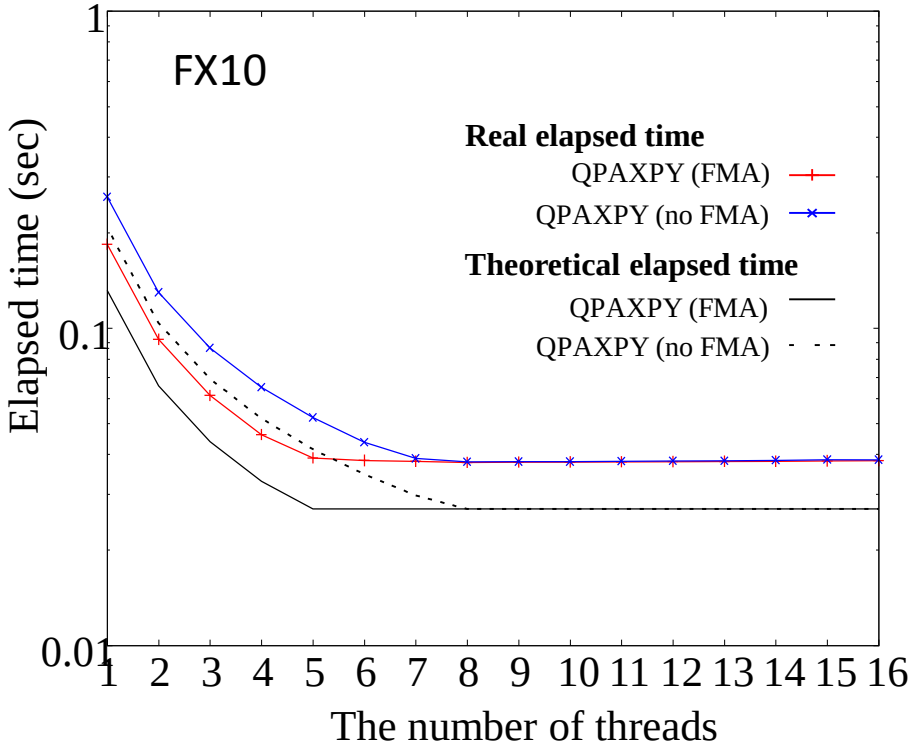
- The ratio of the theoretical elapsed time to the real one is 50% on ICE X and 70% on FX10.
- When FMA instruction is utilized, the elapsed time decreases by about 45% on ICE X and 25% on FX10.
- Since all data are stored on cache, the memory bandwidth does not influence the elapsed time. Therefore, as the number of the threads increases, the elapsed time reduces.

Elapsed time of Level 1 (large size)

AXPY

N=51,200,000





- On small threads, the routne+FMA is faster than the original one.

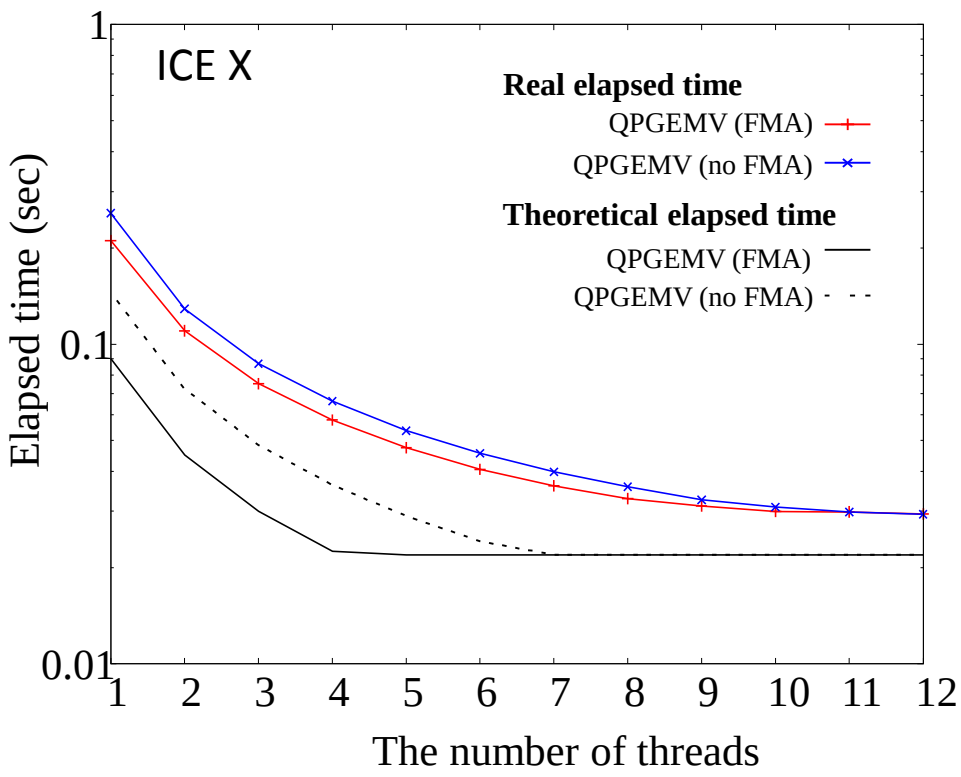
- On many threads, whether FMA is utilized or not, the elapsed times are almost the same.
- At this time, the elapsed time of quad-precision routine is about twice as that of the double-precision one.

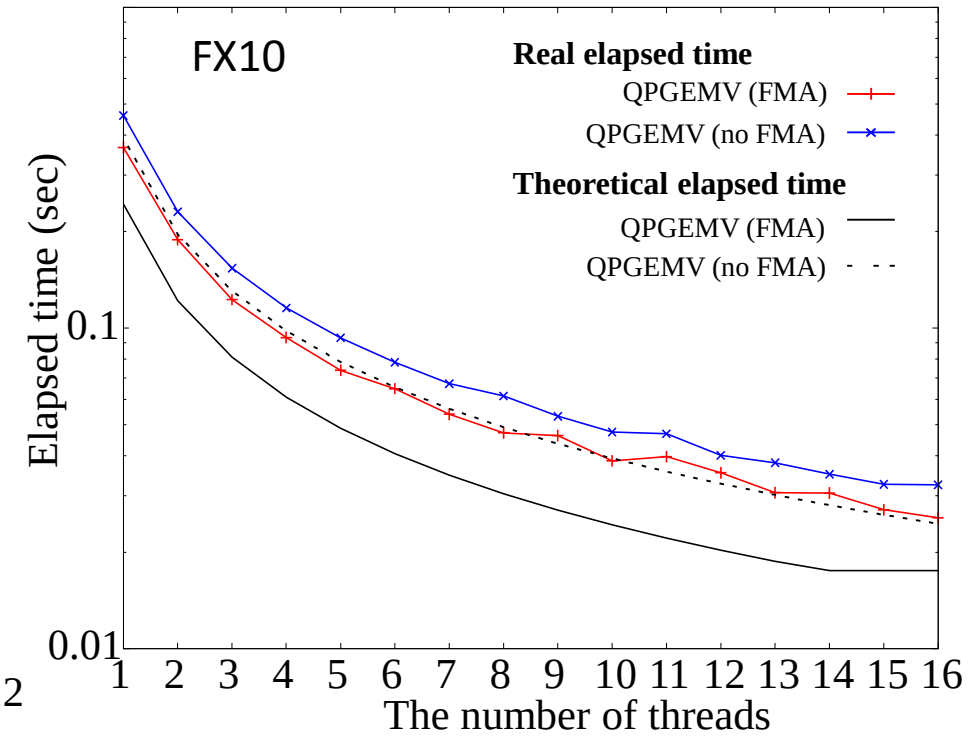
The elapsed time depends on the number of the transferred data from memory.

Elapsed time of Level 2

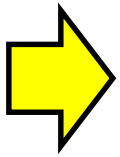
GEMV (Level 2)

N=M=10,000





ICE X achieves speedup up to 9 threads.
Fx10 achieves speedup using 16 threads.



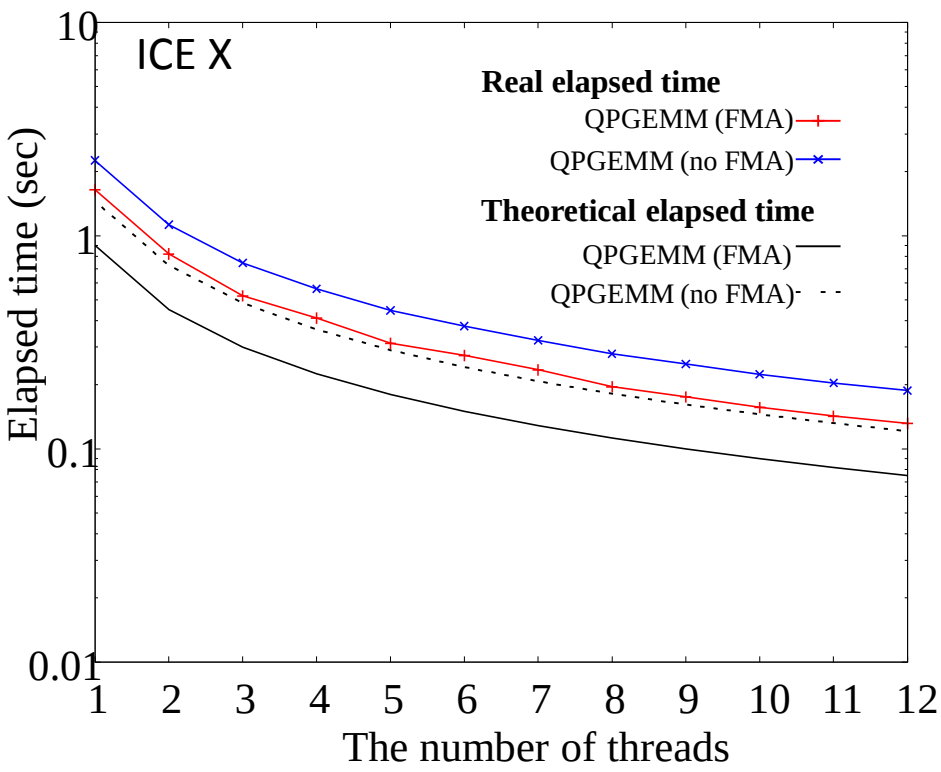
The trend of the speedup is almost the same as the theoretical ones.

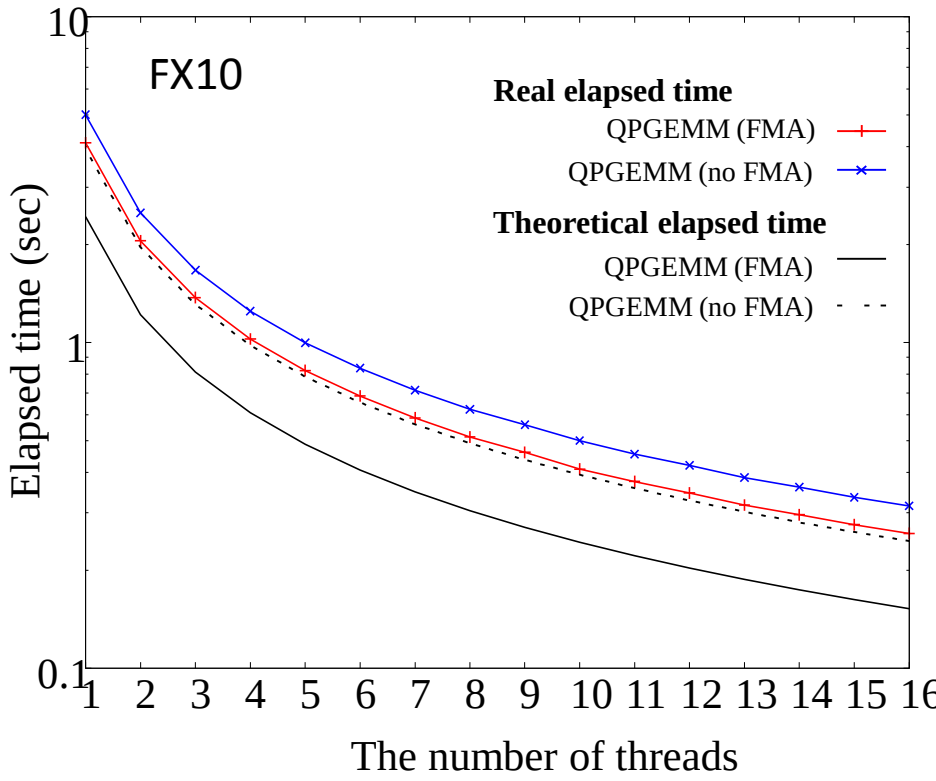
When the number of threads becomes larger, the performance is limited by the memory bandwidth.

Elapsed time of Level3

GEMM (Level3)

N=M=1,000

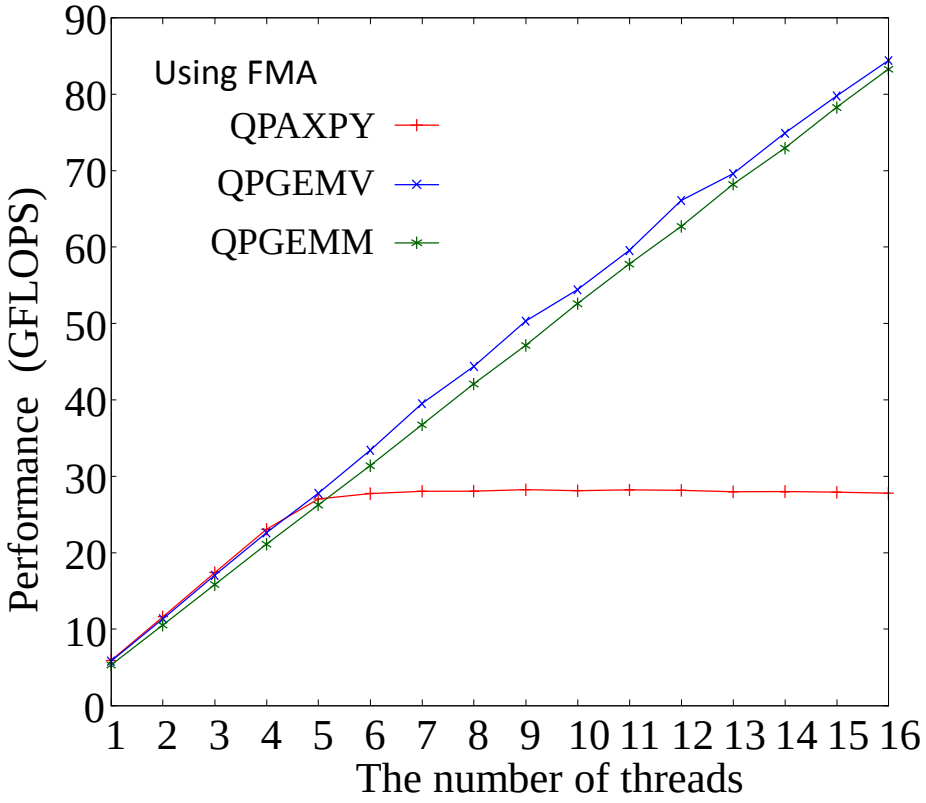


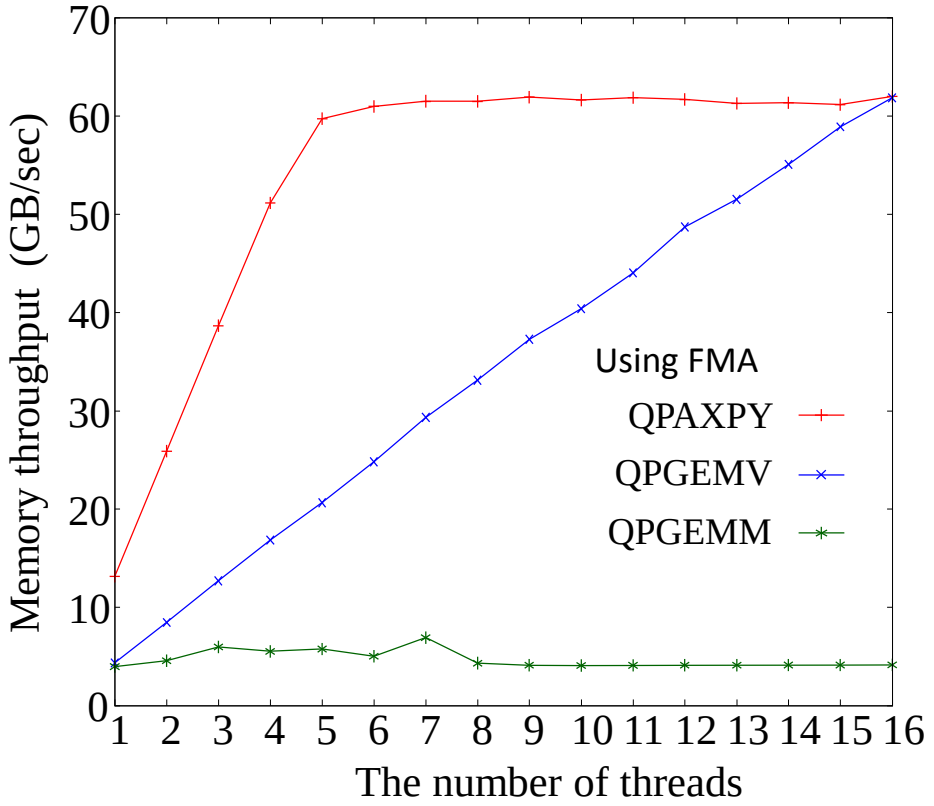


- When using FMA instruction, we obtain 20～30% speedup with any number of threads.
- The ratio of floating-point operations to data movement from memory is sufficiently large for Level 3 routine.

Elapsed time for the Level 3 routine depends on FLOPS, not on the memory bandwidth.

Performance and memory throughput on FX10





The memory throughput of Level 1 routine QPAXPY is saturated with 5 threads.

➡ The performance increase up to 5 threads.

The throughput for Level 2 routine is almost saturated with 16 threads.

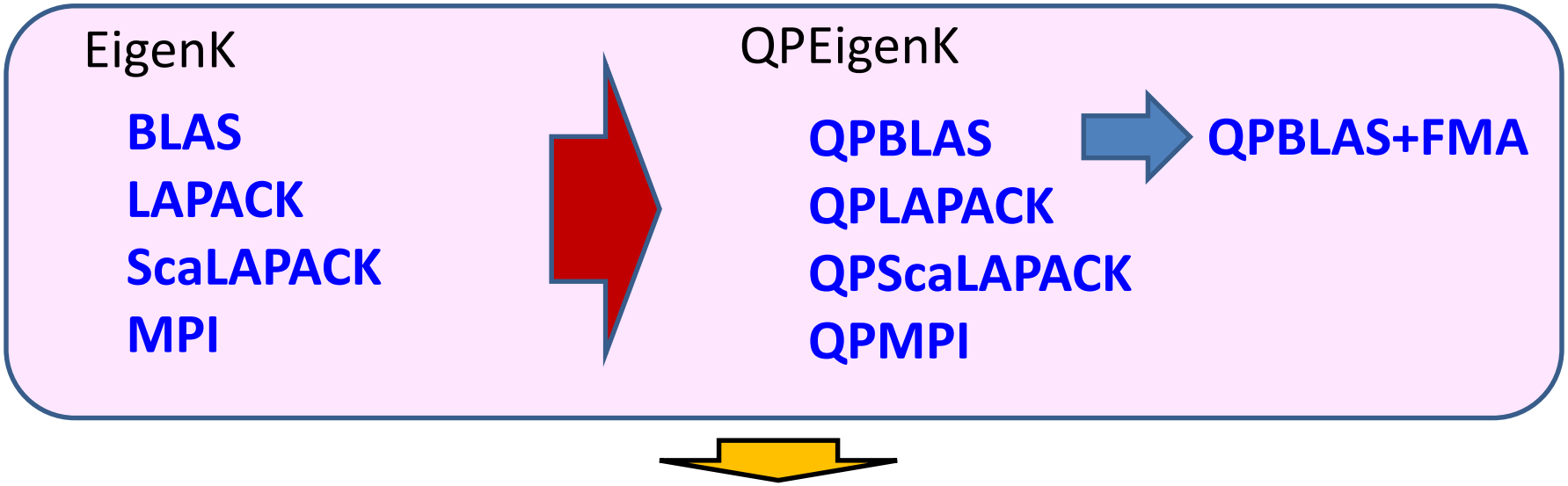
➡ The performance is almost saturated with 16 threads.

When the number of the threads increases for Level 3 , the throughput hardly changed.

➡ The routine has a possibility to achieve a high performance on a many-core processor.

Overview of QPEigenK

We developed the quad-precision eigenvalue solver QPEigenK based on double-precision direct solver EigenK with the double-double (DD) arithmetic. We extended the necessary routines in BLAS, LAPACK, ScaLAPACK, and MPI into quad-precision ones with DD arithmetic.

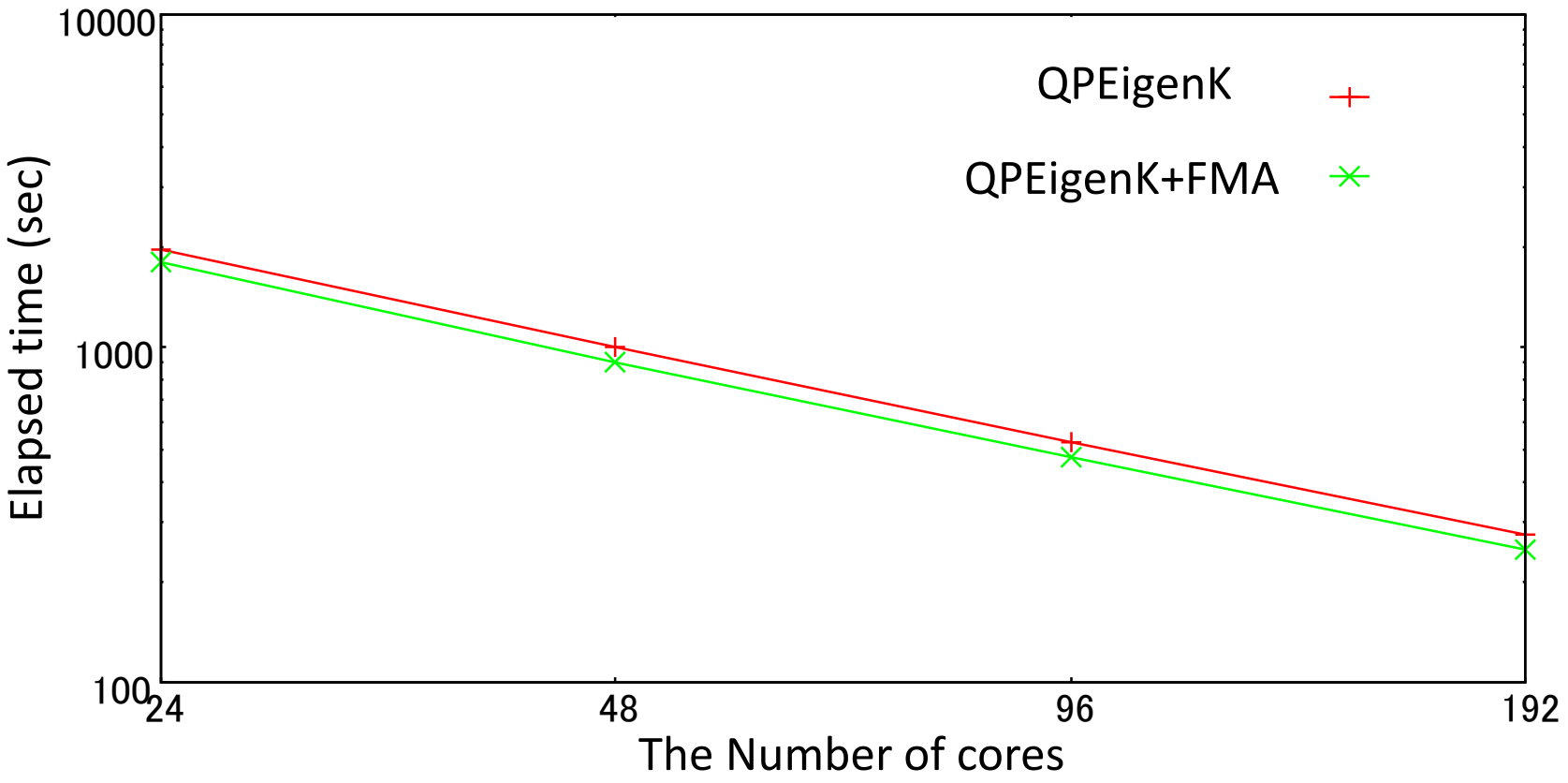


Performance test for QPEigenK with QPBLAS or QPBLAS+FMA.

Parallelization : Flat MPI

Matrix size : 16,000

QPEigenK on ICEX

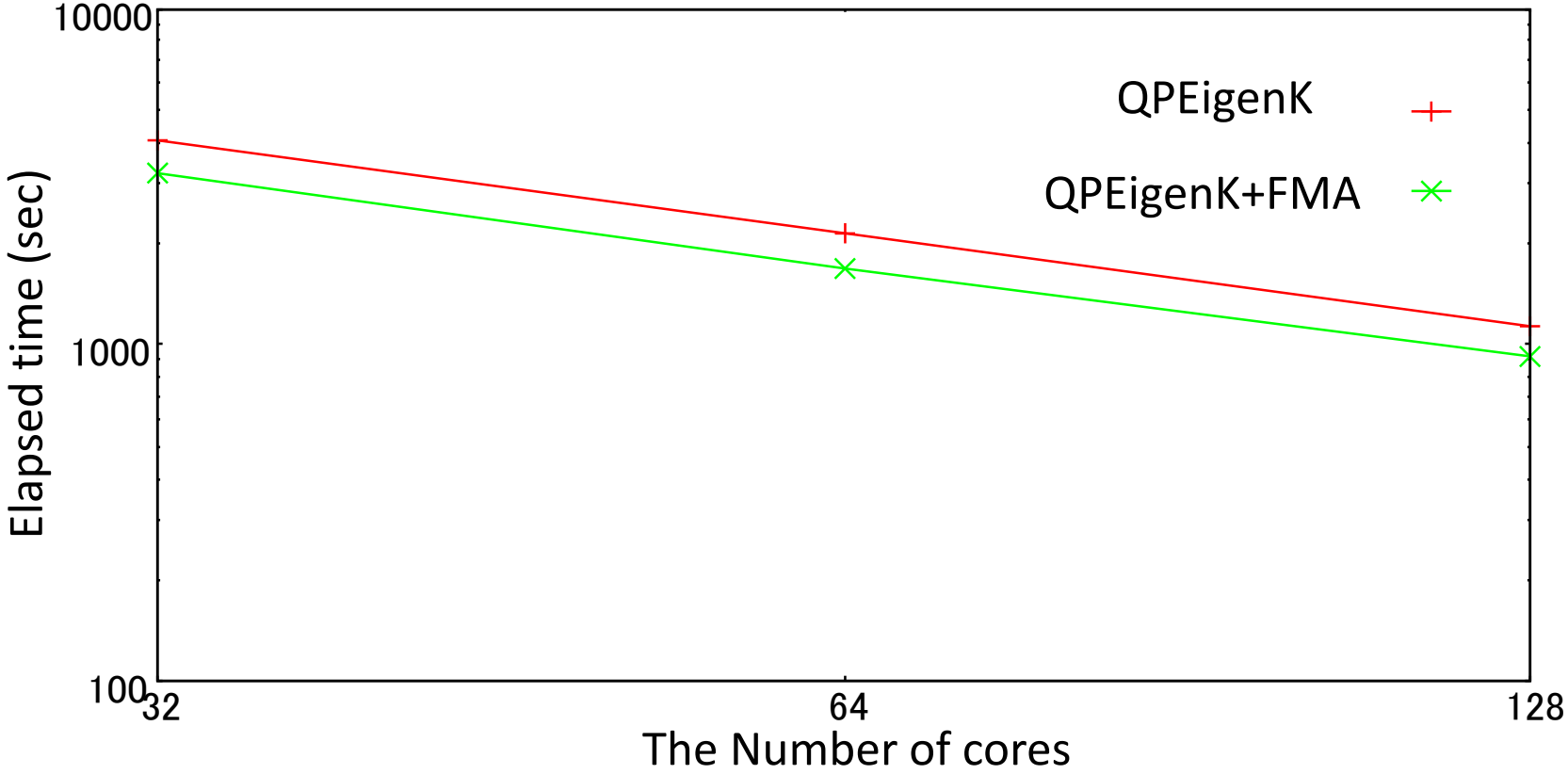


	24 cores			48 cores			96 cores			192 cores		
	QPBLAS	+FMA	ratio	QPBLAS	+FMA	ratio	QPBLAS	+FMA	ratio	QPBLAS	+FMA	ratio
Trd	673.2804	634.8092	0.94	348.4344	328.4325	0.94	186.9448	176.9431	0.95	103.9656	98.85356	0.95
D&C	161.3854	118.9403	0.74	89.03776	64.72398	0.73	48.00655	36.2905	0.76	27.36227	21.63406	0.79
BTr	1148.437	1045.68	0.91	571.3249	517.7198	0.91	285.6589	256.2367	0.90	145.9618	128.2119	0.88
Totel	1983.103	1799.429	0.91	1008.797	910.8763	0.90	520.6103	469.4703	0.90	277.2896	248.6995	0.90

Performance of D&C improves most, when using FMA instruction. But the ratio of D&C is small.

10% speedup

QPEigenK on FX10



	32 cores			64 cores			128 cores		
	QPBLAS	+FMA	ratio	QPBLAS	+FMA	ratio	QPBLAS	+FMA	ratio
Trd	1140.561	1113.705	0.98	586.678	567.0671	0.97	332.7886	326.2054	0.98
D&C	248.6933	222.5274	0.89	145.4987	131.6127	0.90	78.28661	71.75055	0.92
BTr	2689.077	1902.329	0.71	1394.92	997.367	0.71	713.5748	523.217	0.73
Totel	4078.332	3238.561	0.79	2127.097	1696.047	0.80	1124.65	921.173	0.82

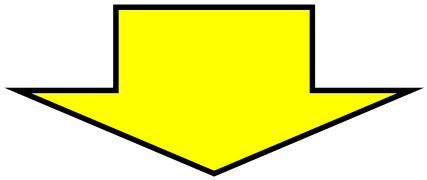
The decrease in the elapsed time of back-transformation is about 30%.

Total:20% speedup

Performance improvement by tuning

QPBLAS and QPBLAS+FMA are programed without special tunings.

An appropriate tuning has a possibility to improve their performance.



In order to examine the efficiency of loop unrolling for QPBLAS+FMA, we unroll the loops of quad-precision GEMV manually and evaluate the performance.

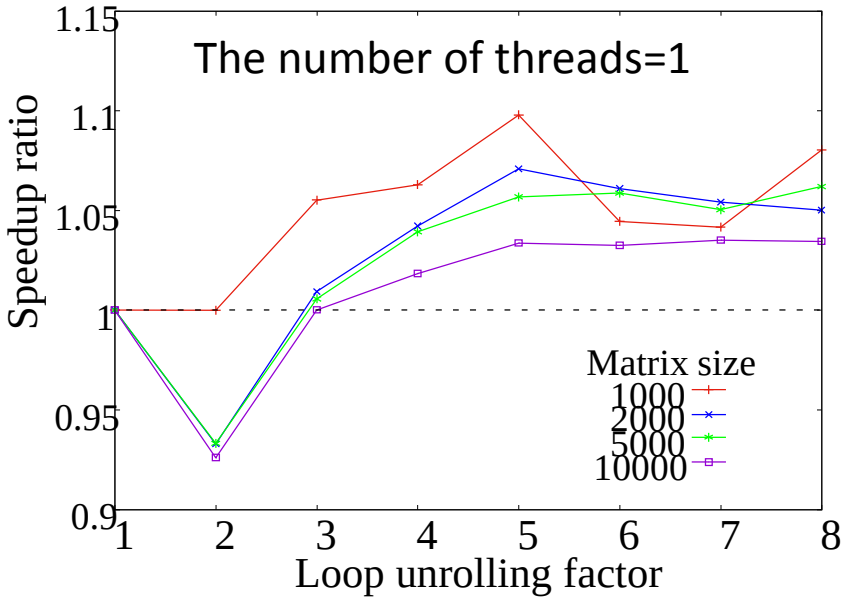
Unrolling strategy for QPGEMV

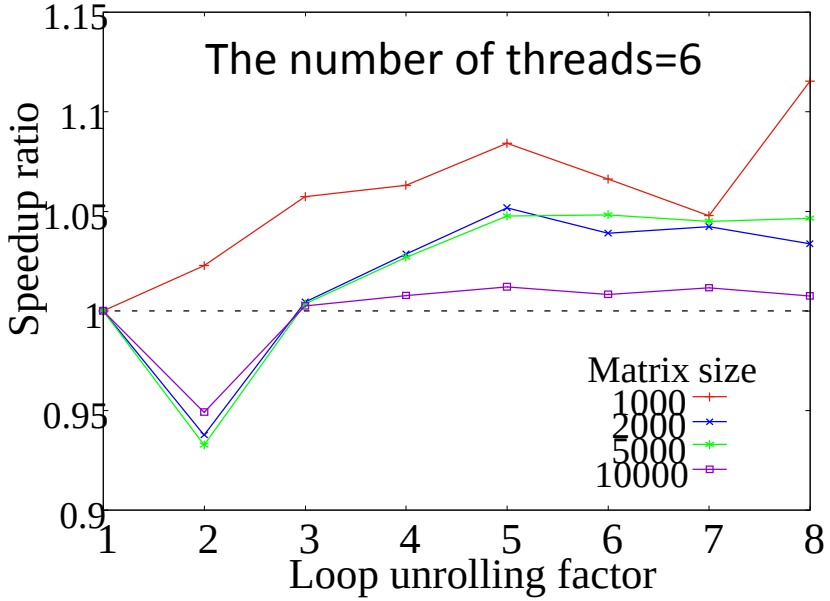
All calculations are executed with quad-precision using DD arithmetic.

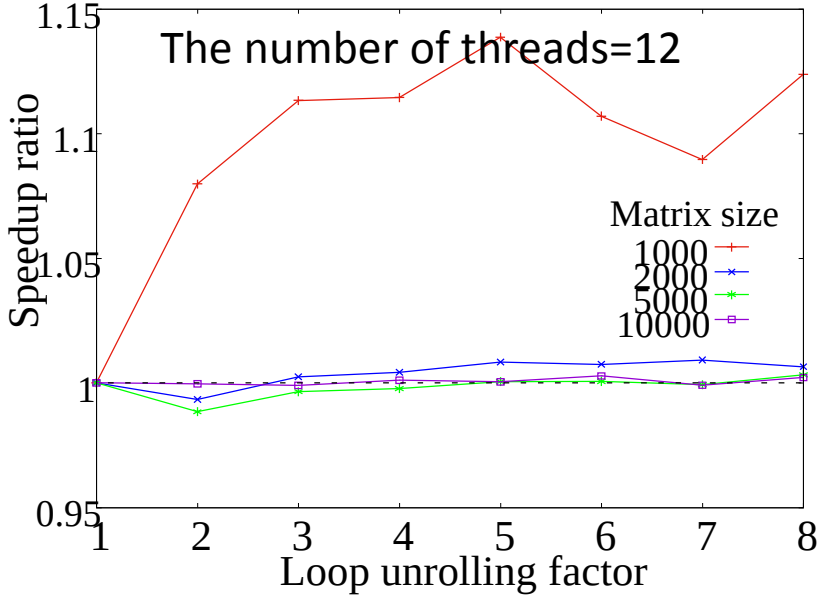
```
#pragma omp parallel for
for (j=0, j<M, i++){
  y[j]=0.0;
}
#pragma omp parallel for
{
  for (i=0,i<N,j++) {/* unrolling */
    x[i]=alpha*x[i]; /* SIMD */
  }
#pragma omp parallel private (work,j)
  for (j=0,j<M,j++){
    work[j]=0.0;
  }
#pragma omp for
  for (i=0,i<N,i++){
    for (j=0,j<M,j++){ /* unrolling */
      work[j]=work[j]+A[i][j]*x[i]; /* SIMD */
    }
  }
#pragma omp critical
  for (j=0,j<M,i++){
    y[j]=y[j]+work[j]; /* SIMD */
  }
}
```

Reduction operation of OpenMP is not applicable for Bailey's arithmetic.

Performance improvement by loop unrolling on ICE X

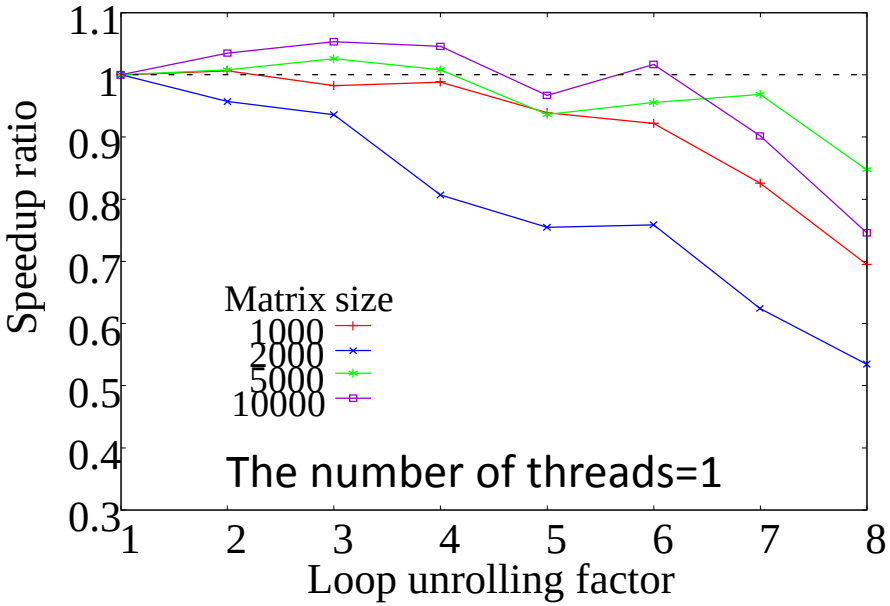


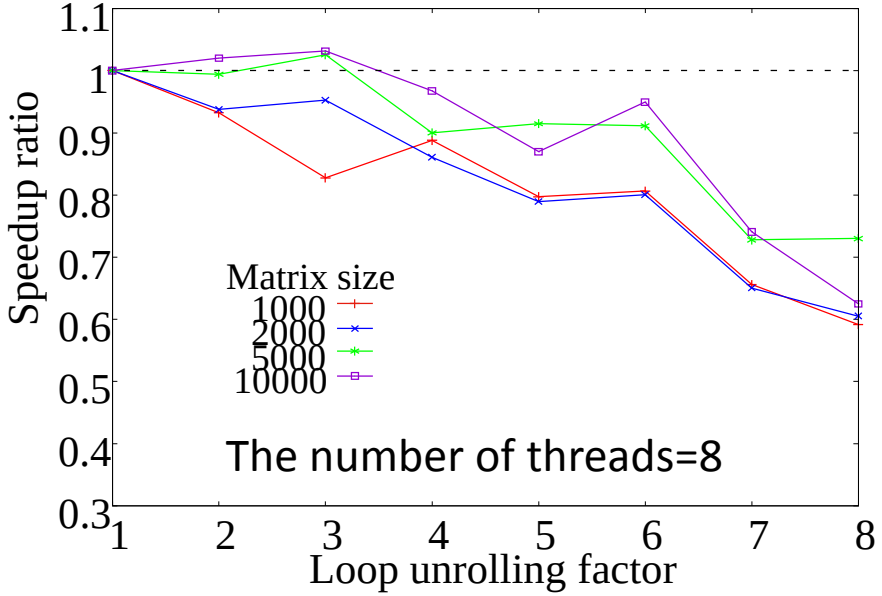


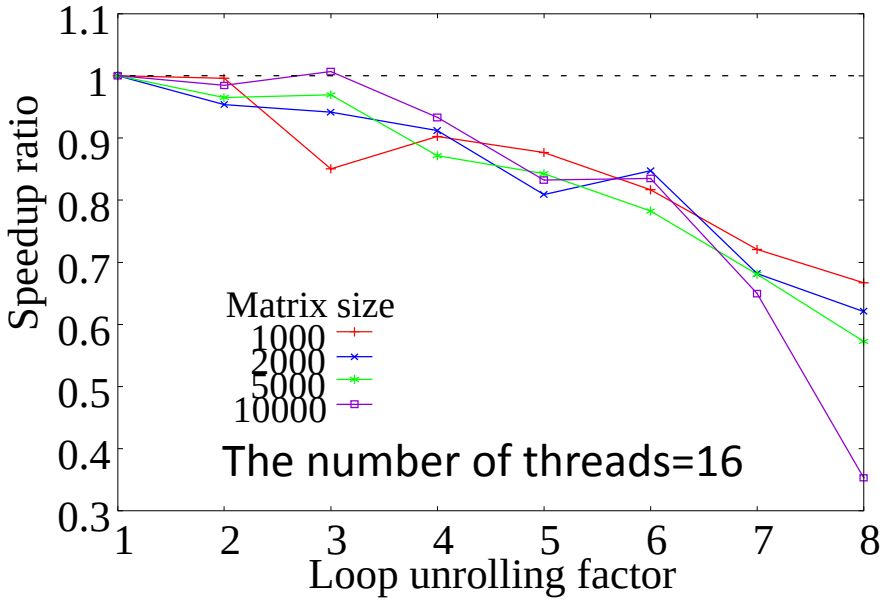


- When the matrix size is 1000, all date can be stored on cache. Therefore, speed up is obtained by unrolling.
- For other matrix size cases, the unrolling realized speedup up to 6 threads.
- When the number is 12, the performance depends on the memory bandwidth except for 1000-dimensional case. Therefore, unrolling has little effect for speedup.

Performance improvement by loop unrolling on FX10

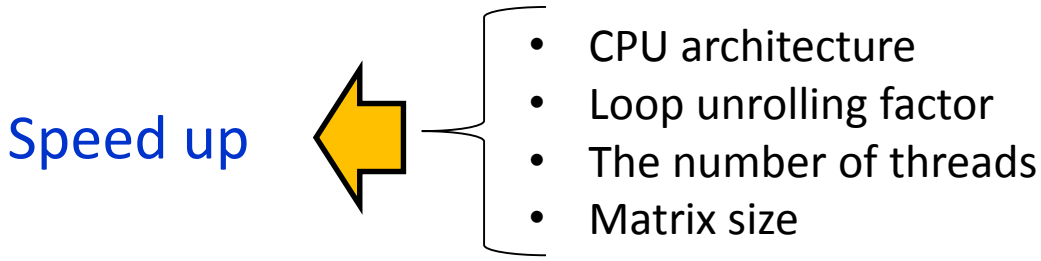






- For large matrix, we can obtain speedup by unrolling.
- In general, unrolling gave bad effect for speedup.

Performance improvement by loop unrolling



In this case, we unrolled the Level 2 routine.

- Two-level nested loop
- Unrolled only Inner loop

When we unroll the outer loop, we have a possibility to obtain more speedup.

Level 3 routines ⇨ Three-level nested loop

The combination of the loop unrolling for each loop is complicated.

It is difficult to find the optimal unrolling factor. ⇨ **Automatic performance tuning is the most practical strategy.**

Conclusions

We developed the quadrature precision BLAS (QPBLAS) using FMA instruction.

QPBLAS with FMA is basically faster than the original QPBLAS. However, the number of the threads is large, their performances are the almost same for some Level 1 and level 2 routines.

For the quadrature precision eigenvalue solver (QPEigenK), about 10% (ICEX) or 20% (FX10) speedup can be obtained by only replacing the original QPBLAS to QPBLAS+FMA.

For the speedup for the double-double arithmetic, FMA instruction is very efficient.

QPBLAS+FMA has a possibility of speedup using loop unrolling. But, it is difficult to find the optimal unrolling factor.

Auto performance tuning is the most practical strategy to find the optimal value.