# 多倍長精度基本線形計算の高速化とその応用


静岡理工科大学
幸谷智紀
kouya.tomonori@sist.ac.jp

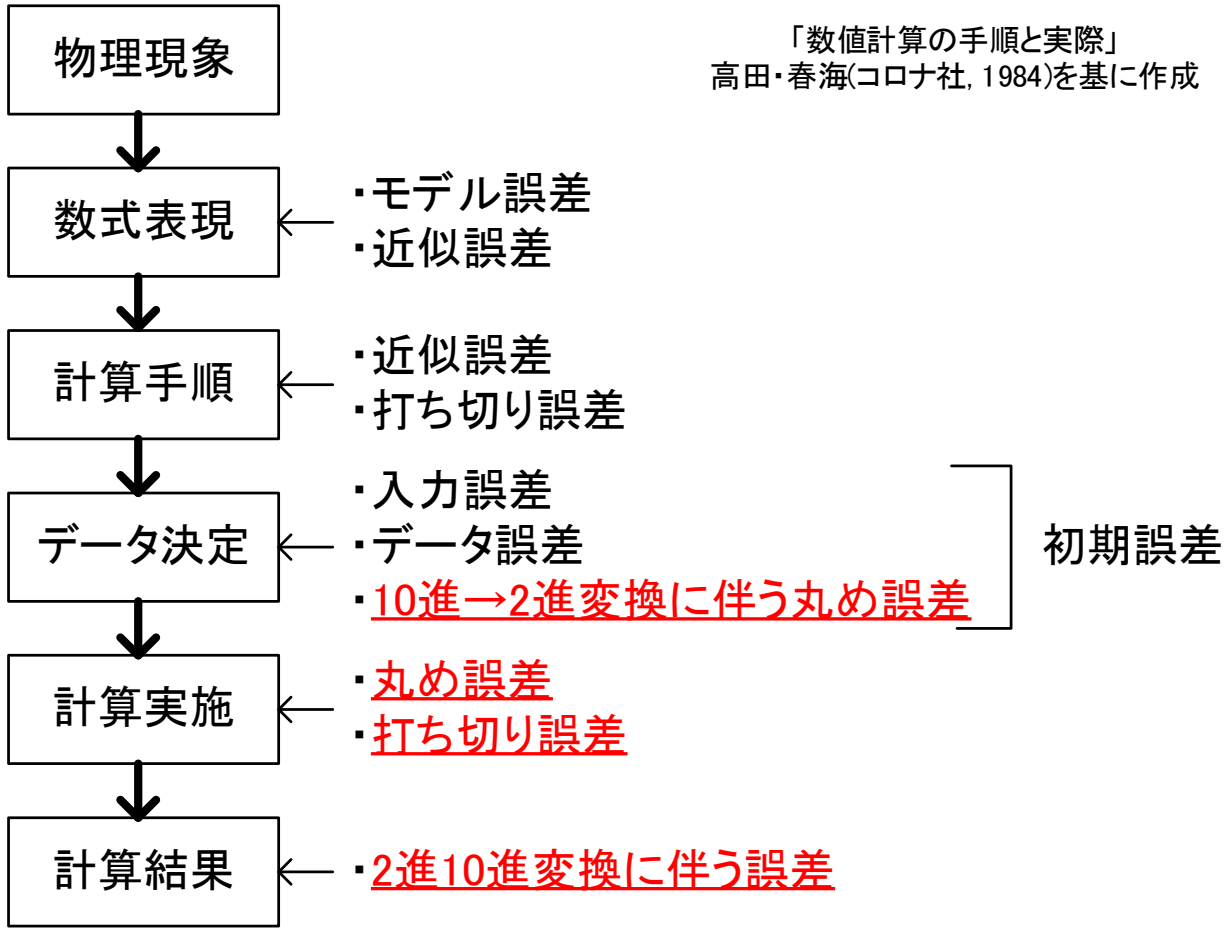
第20回HPC-Physワークショップ
2023年12月8日（金）

# 講演概要

I. 多倍長精度浮動小数点計算の目的
II. 基本線形計算の最適化・行列乗算（xGEMM)を中心に
III. 応用事例
    A) 混合精度反復改良法
    B) 陰的Runge-Kutta法の高速化（昔の話題）
    C) 悪条件代数方程式の求解
IV. 今後の課題

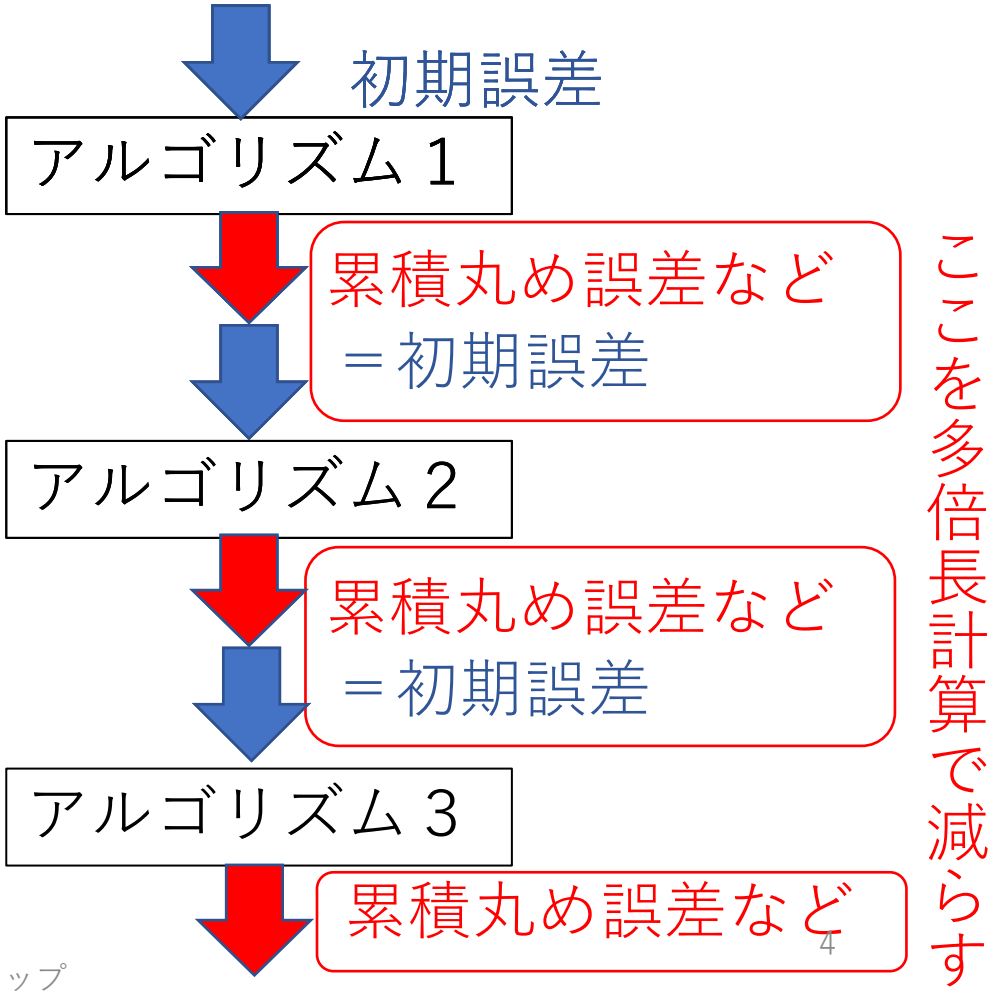
# I. 多倍長精度浮動小数点計算の目的

# 多倍長精度数値計算の目的

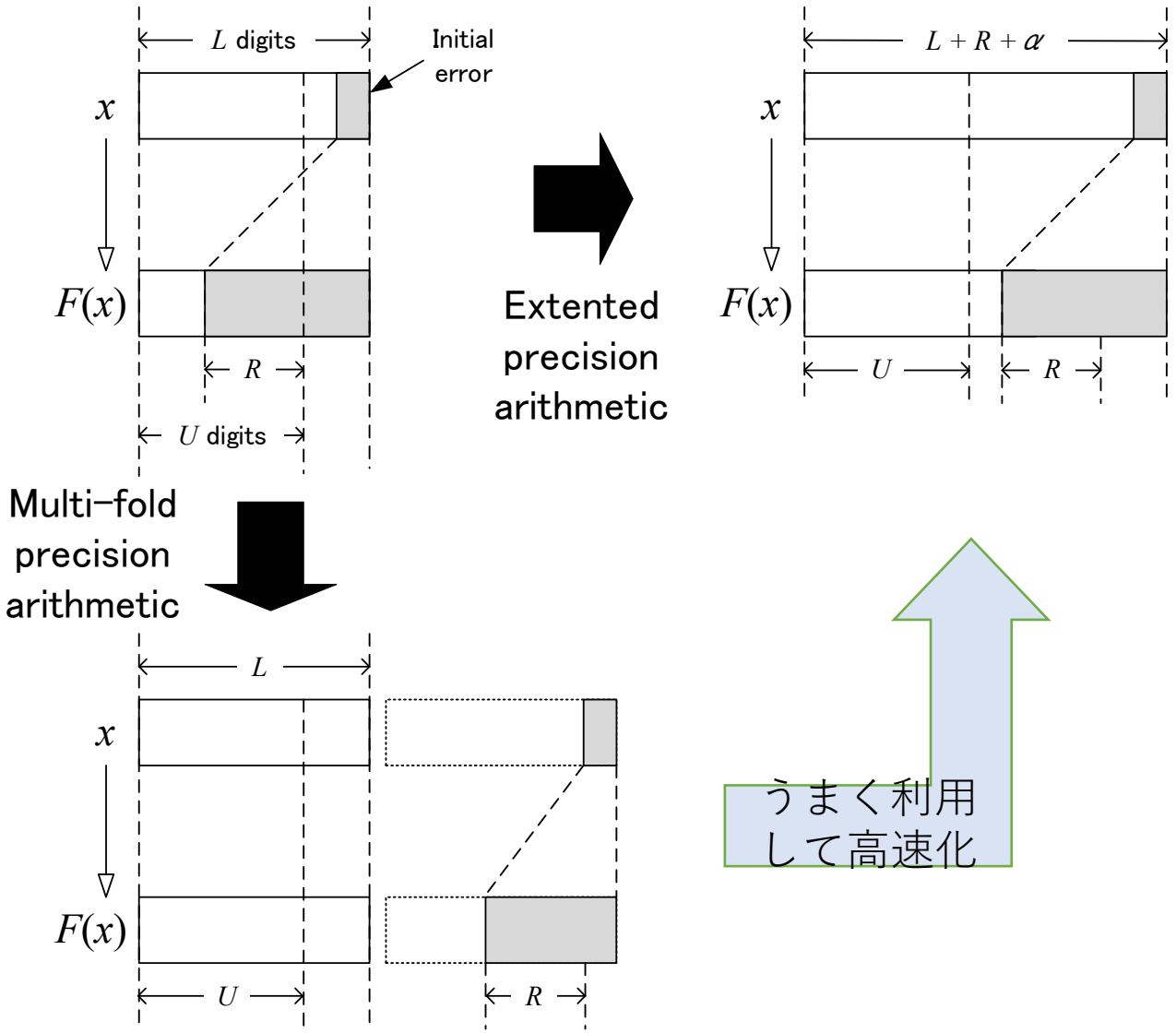



「数値計算の手順と実際」
高田・春海(コロナ社, 1984)を基に作成

初期誤差→入力値に含まれて次のアルゴリズムへ引き継がれる



# なぜ多倍長精度演算ライブラリが必要か？




- **多倍長精度浮動小数点演算**：丸めによる初期誤差・計算中に生じる誤差の影響を仮数部の桁数を増やすことで減じる
- **多重精度演算→再現可能計算(Reproducible)**：上記から再正規化の手間を減らして効率化を図る

→現状ではどちらも必要

Ex: 常微分方程式の補外計算(Arith2019)

1. DD精度で初期系列計算
2. 補外計算に2重精度計算

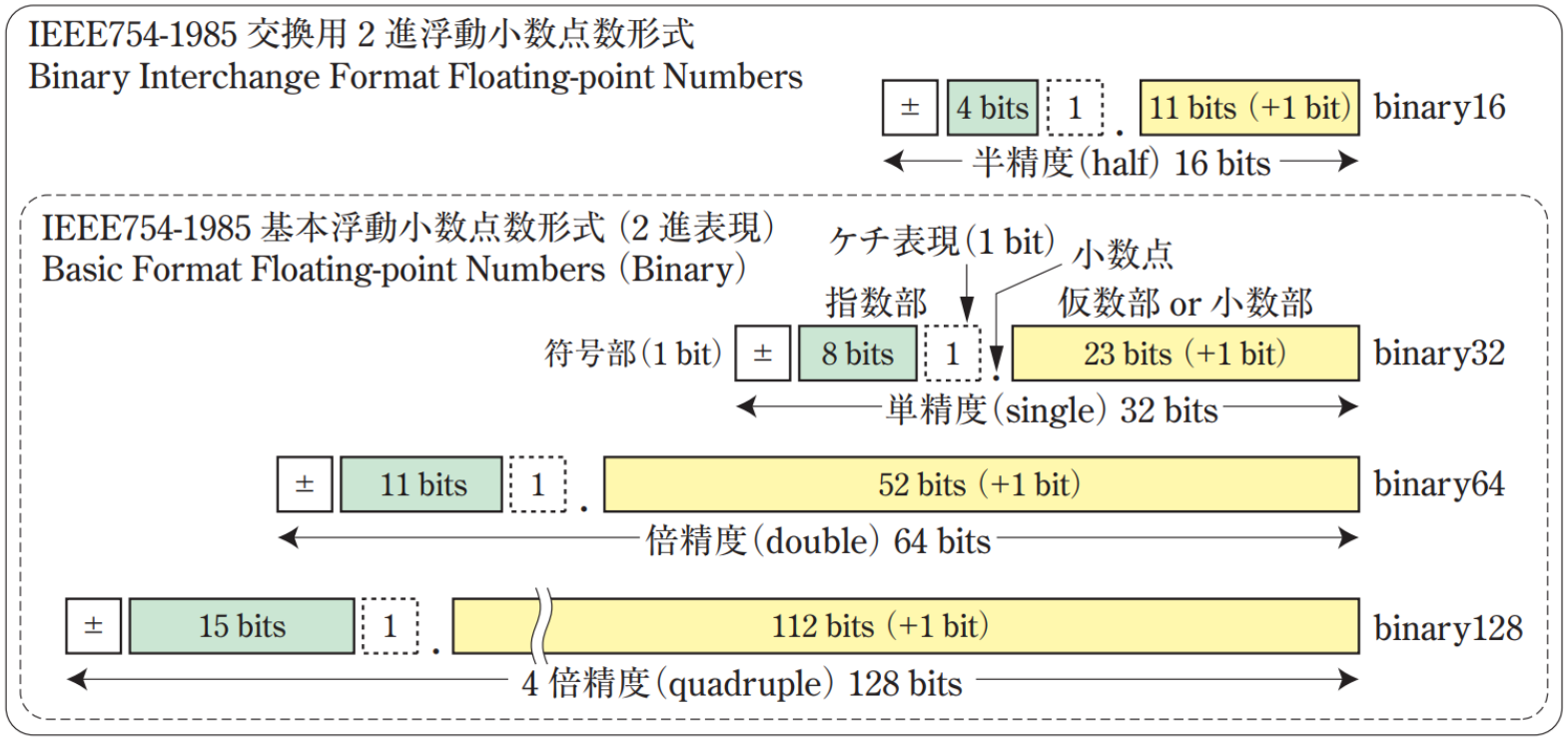
# 浮動小数点数形式

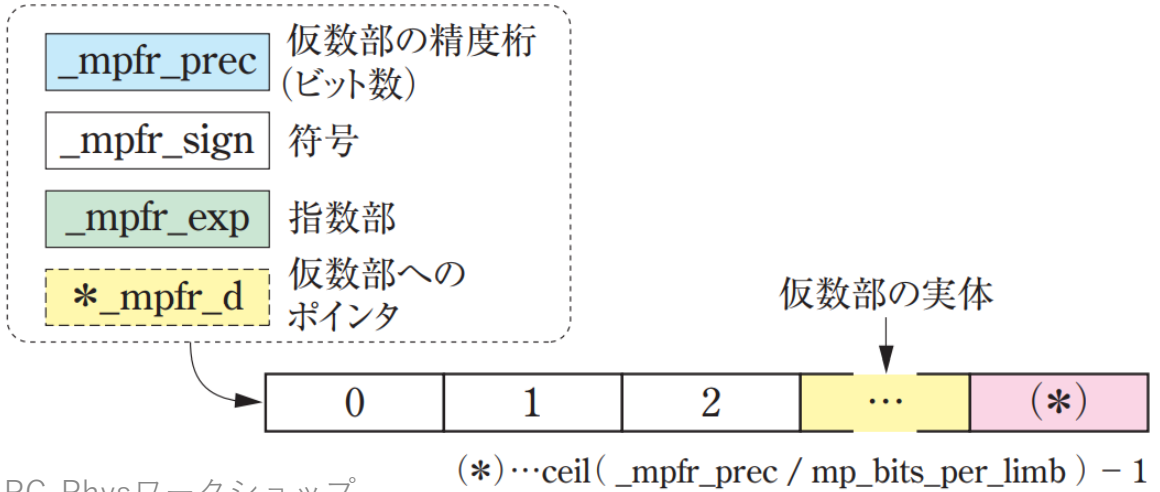
ハードウェアサポートのある標準浮動小数点数

- IEEE754 binary16(半精度), binary32(単精度), binary64(倍精度)

ソフトウェアで実装する多倍長精度浮動小数点数

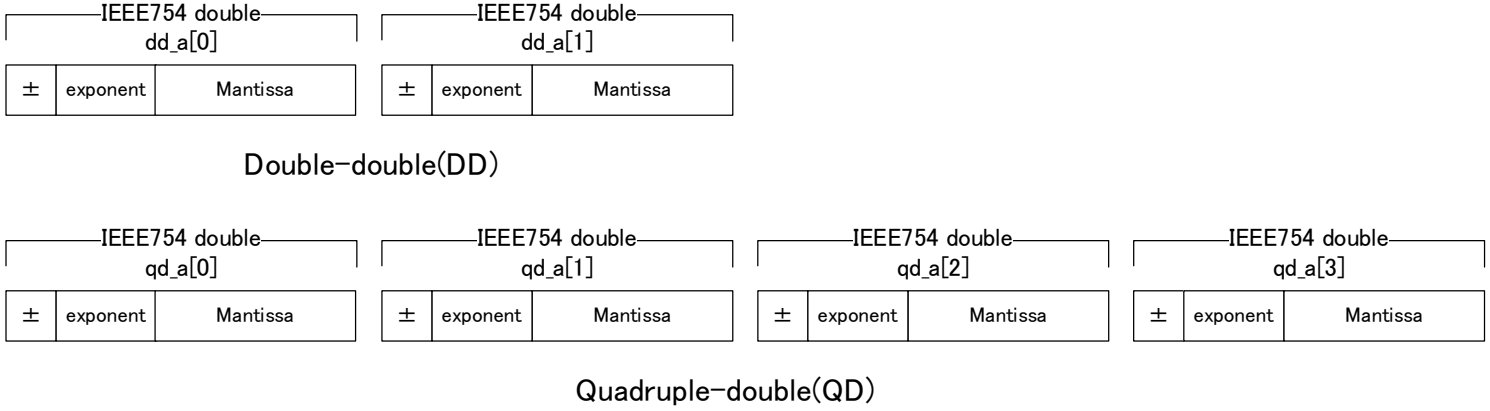
- Double-double(DD), Triple-double(TD), Quadruple-double(QD)(, _float128(binary128))
- MPFR(/GMP)



# 5．無誤差変換技法とQD

- Dekker→double-double(DD)→Quadruple-Double(QD)




- 無誤差変換技法の積み重ねによるDD，QD演算の実装

**Algorithm 4** QuickTwoSum

$(s, e) := \mathrm{QuickTwoSum}(a, b)$

$s := a \oplus b$

$e := b \ominus (s \ominus a)$

**return** $(s, e)$

**Algorithm 7** $(p, e) := \mathrm{TwoProd\text{-}FMA}(a, b)$

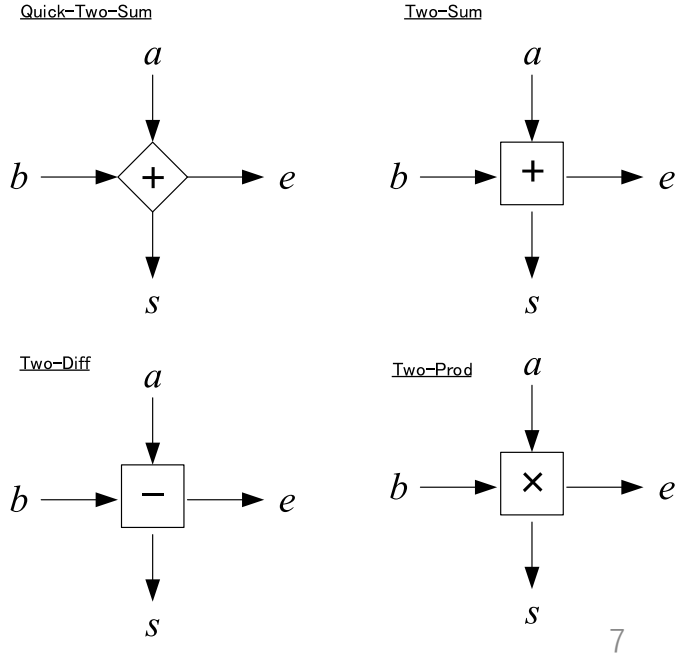
$p := a \otimes b$

$e := \mathrm{FMA}(a, b, -p)\ (= a \times b - p)$

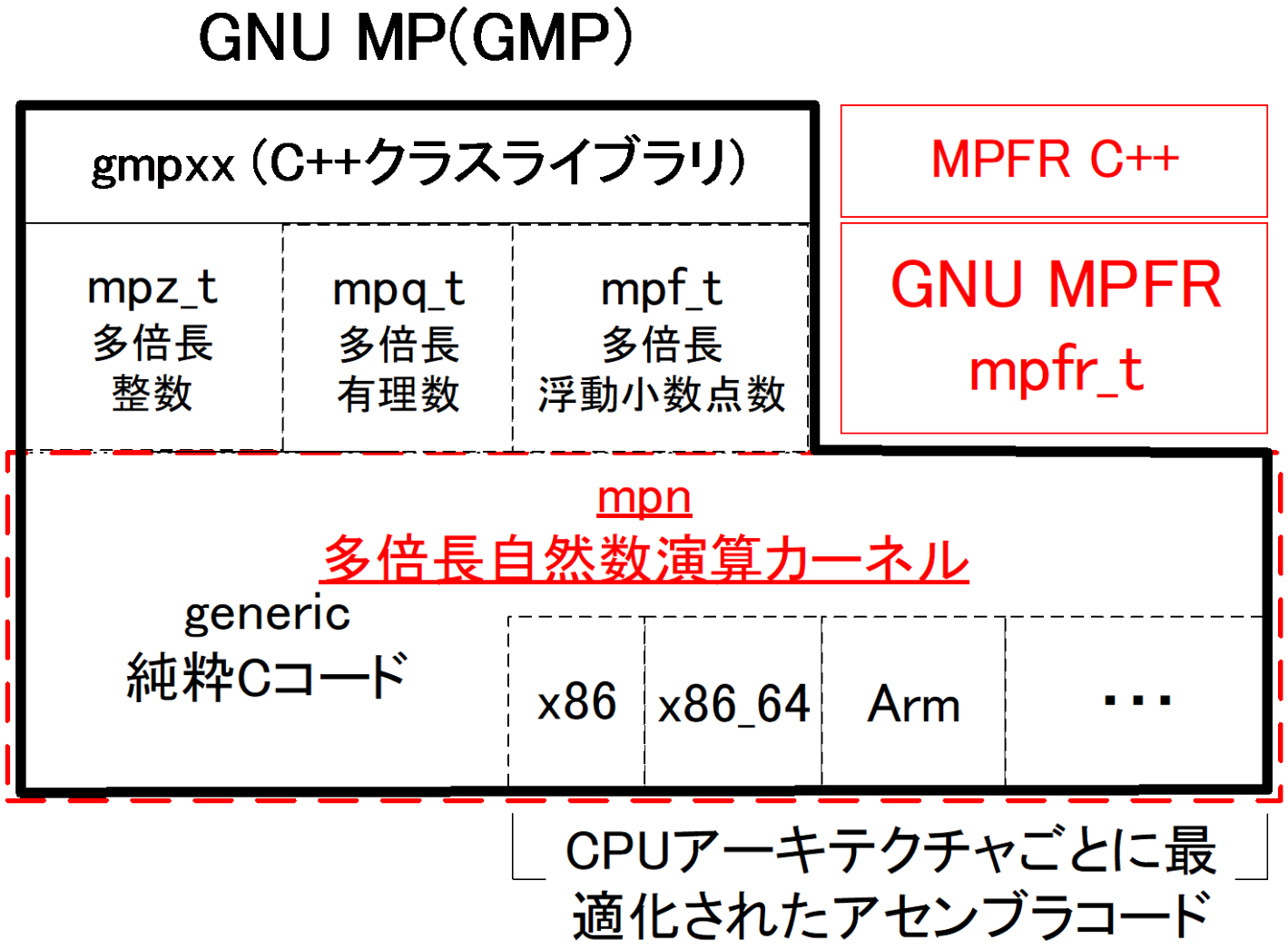
$\mathrm{return}(p, e)$

**Algorithm 5** TwoSum と TwoDiff

| $(s, e) := \mathrm{TwoSum}(a, b)$ | $(d, e) := \mathrm{TwoDiff}(a, b)$ |
|---|---|
| $s := a \oplus b$ | $d := a \ominus b$ |
| $v := s \ominus a$ | $v := d \ominus a$ |
| $e := (a \ominus (s \ominus v)) \oplus (b \ominus v)$ | $e := (a \ominus (d \ominus v)) \ominus (b \oplus v)$ |
| $\mathbf{return}(s, e)$ | $\mathbf{return}(d, e)$ |



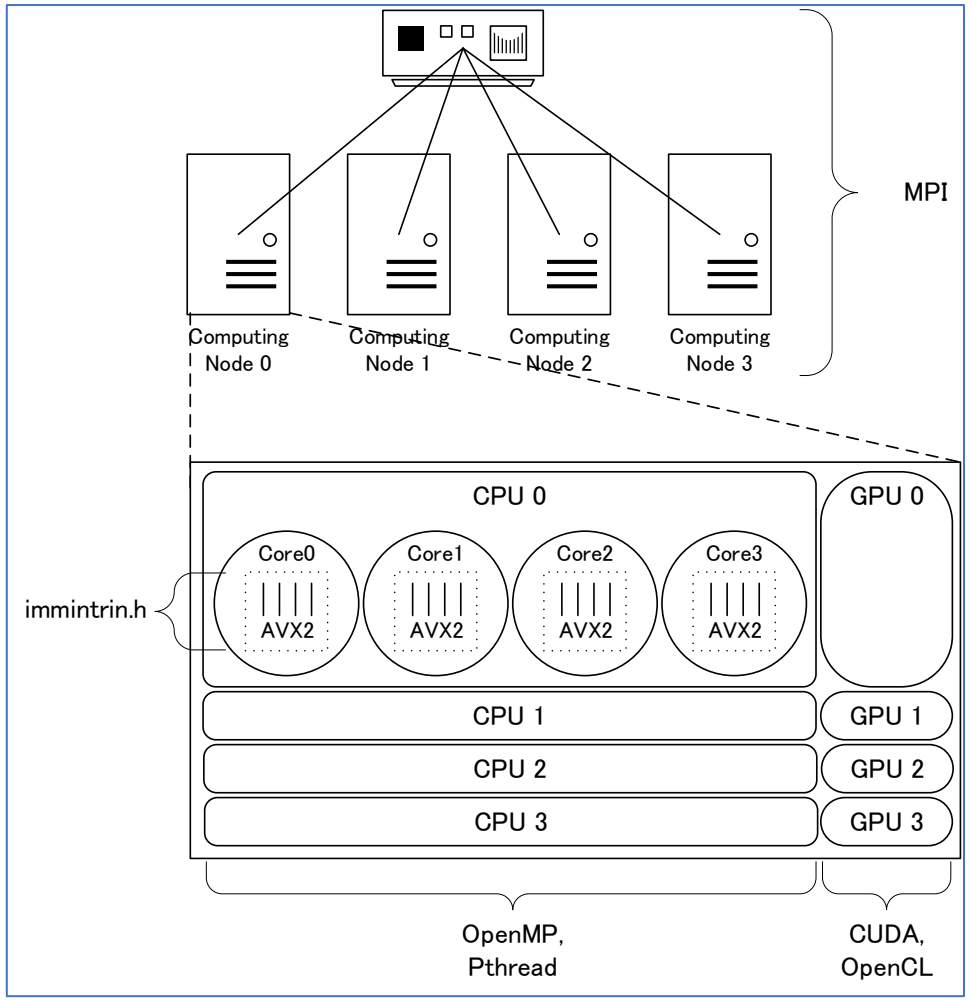
# 3．GNU MP(GMP)の高速性




- 最適なアルゴリズムの選択
- CPUアーキテクチャごとに最適化されたアセンブラルーチンの採用
- T.Granlundの変質狂的な最適化職人芸の賜物
- 複雑すぎて，mini-gmpにまとめ直す動きも

# Categorization of optimization techniques for HPC

## Architecture-oriented speedup technics




## Optimization using computing algorithm

Long-digit real multiplication
- Karatsuba method
- Toom-Cook method
- FFT

Complex multiplication
- 3M method

Matrix multiplication
- Blocking(Tiling) algorithm
- Strassen and Winograd
- Ozaki scheme (New!)

# II. 基本線形計算の最適化
# ・行列乗算（xGEMM)を中心に

# Optimization of multiple-precision matrix multiplication and its application

| xGEMM | CPU | | | | GPU | |
|---|---|---|---|---|---|---|
| Opt.Method | None | AVX2 | OpenMP | Ozaki Scheme | CUDA | Ozaki Scheme |
| DS | ? | ? | ? | ? | Mukunoki | Mukunoki |
| TS | ? | ? | ? | Utsugiri | Utsugiri | Utsugiri |
| QS | ? | ? | ? | ? | ? | ? |
| IEEE754 Binary128 | MPBLAS | ? | MPBLAS | Mukunoki | Mukunoki | Mukunoki |
| DD | MPBLAS, BNCmatmul | Lis, MuPAT BNCmatmul | Lis,MuPAT, MPBLAS, BNCmatmul | Utsugiri | Mukunoki | Mukunoki |
| TD | BNCmatmul | BNCmatmul | BNCmatmul | Utsugiri | Utsugiri | Utsugiri |
| QD | MuPAT, MPBLAS, BNCmatmul | BNCmatmul | MuPAT, MPBLAS, BNCmatmul | Utsugiri | ? | ? |
| MPFR | MPBLAS, BNCmatmul | ? | MPBLAS, BNCmatmul | Utsugiri | CUMP | ? |

MPLAPACK/MPBLAS https://github.com/mahonakata/mplapack
MuPAT Yagi, H et.al(2020)
Lis https://www.ssisc.org/lis/
CUMP https://github.com/skystar0227/CUMP
BNCmatmul https://na-inet.jp/na/bnc/
Mukunoki Mukunoki, D et.al.(2021)
Utugiri Utsugiri(2021-2023)

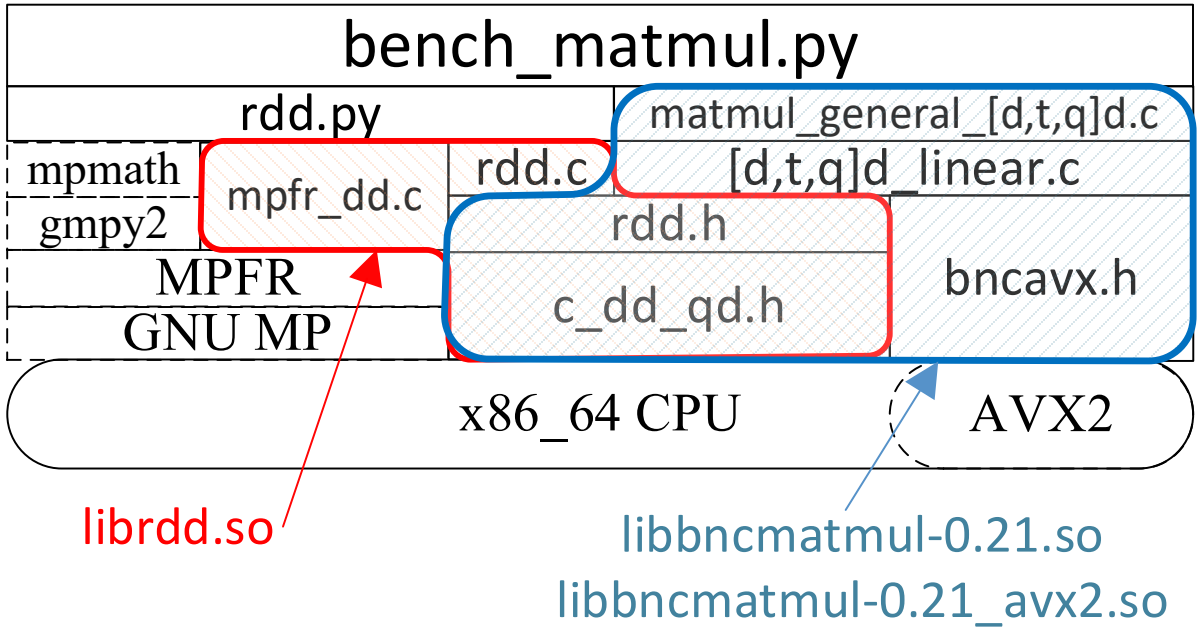
- Completed implementation of optimized DD, TD, QD and MPFR matrix multiplication (xGEMM)
- Found the effectiveness of Ozaki scheme
- Confirmed the effectiveness of Ozaki scheme to LU decomposition

# BNCpackからBNCmatmulへ

- BNCpack: MPFR/GMPベースの多倍長精度数値計算ライブラリ(MPIサポート)

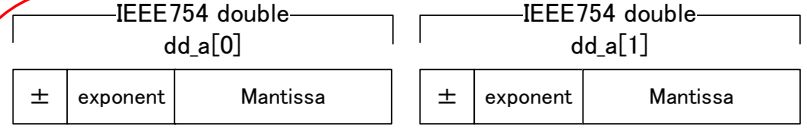
- BNCmatmul: MPFR/GMP, DD, TD, QD精度基本線形計算の最適化ライブラリ https://github.com/tkouya/bncmatmul
  - SIMD(AVX2, AVX-512)のサポート
  - OpenMPによる並列化
  - Strassen, Winograd行列乗算の実装
  - 尾崎スキームの実装
  - LU分解（直接法）
  - DLL化＋Pythonサポート

# BNCmatmul's software layer

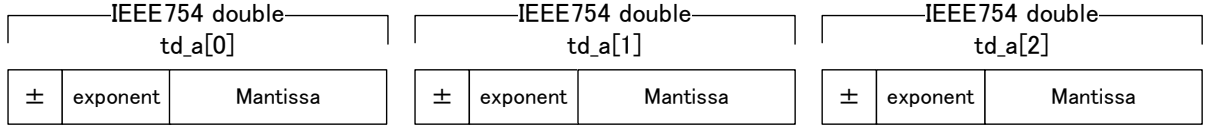



- All FP computation is supported on CPU.
- Limited functionaries but faster than MPLAPACK/MPBLAS
- Multi-component-way fxed-precision arithmetic is constructed with original ANSI C codes : Double, DD, TD, and QD
- Multi-digit-way arbitrary-precision arithmetic is based on MPFR/GMP
- Serial multiple-precision computation is runnable on Python ecosystem.

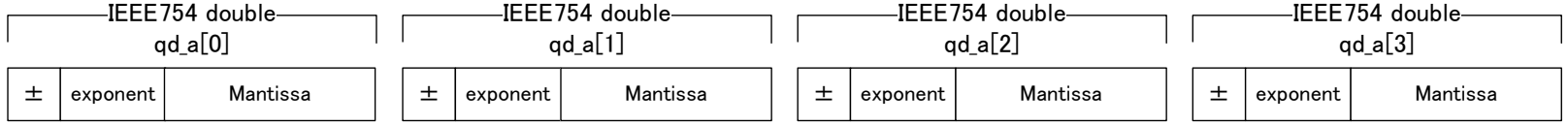
# Our targeting FP precision: DD, TD, QD and MPFR




Double-double(DD)



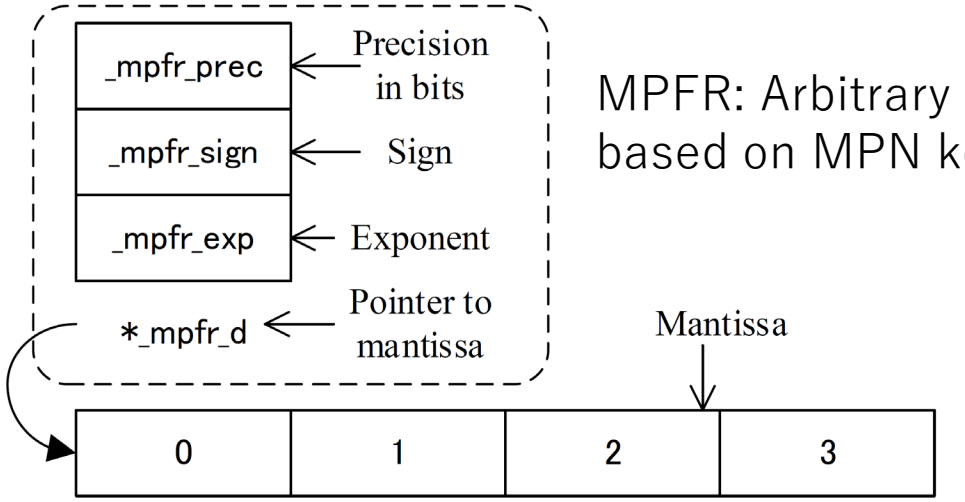

Triple-double(TD)




Quadruple-double(QD)

## Accelerated with AVX2

| | $\oplus$, $\ominus$ | $\otimes$ | FMA | if | Machine epsilon |
|---|---|---|---|---|---|
| DDadd and DDmul | 21 | 3 | 1 | 0 | 1.2e-32 |
| TDadd and TDmul | 80 | 3 | 5 | 8 | 1.4e-48 |
| QDadd and QDmul | 294 | 13 | 10 | 10 | 1.5e-64 |




MPFR: Arbitrary precision FP arithmetic based on MPN kernel of GNU MP

MPBLAS in MPLAPACK: MPREAL C++ class library calls MPFR functions
BNCmatmul: Directly calls MPFR functions→about 10% better performed

# Computing environment for benchmark tests

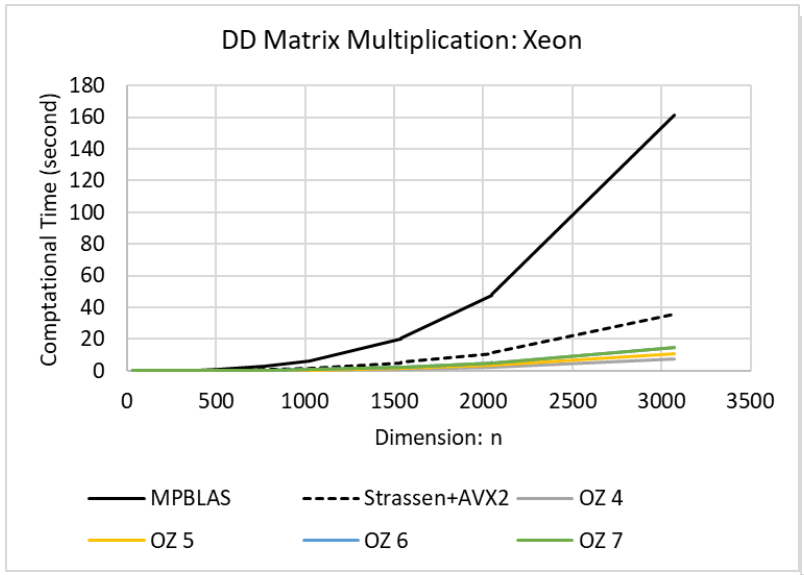
$$A \in \mathbb{R}^{n \times n}, B \in \mathbb{R}^{n \times n} \Rightarrow AB \in \mathbb{R}^{n \times n}$$

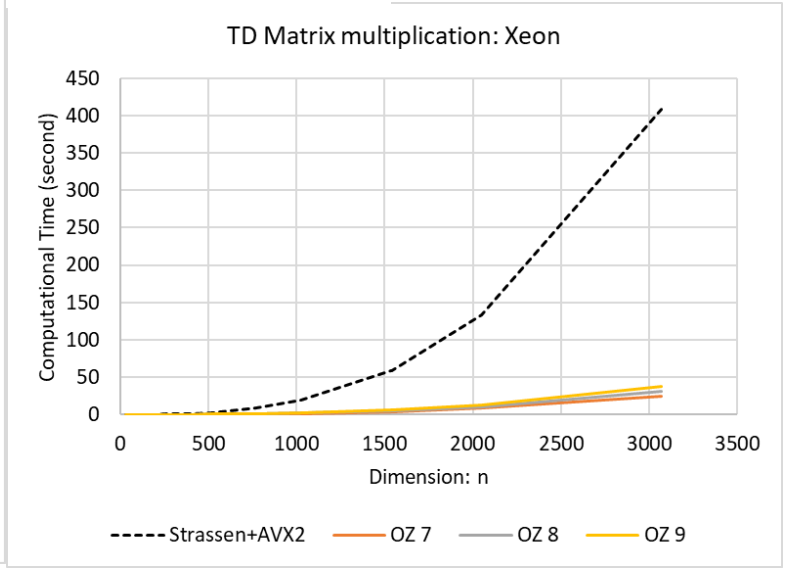
Each element: $(ru - 0.5) \times \exp(rn)$
$ru$: uniform rand on [0, 1], $rn$: normal dist. rand on [-1, 1]

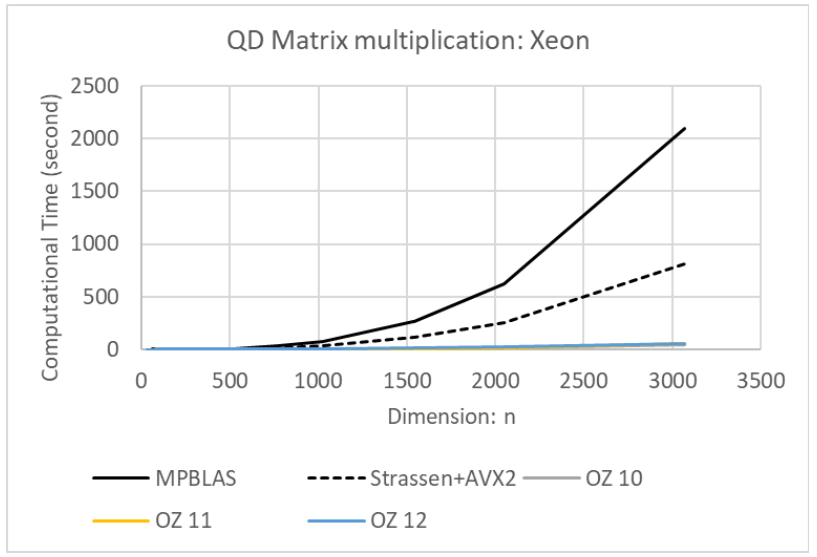
- [Xeon] Intel Xeon W-2295 3.0GHz 18 cores, Ubuntu 20.04.3 LTS, Intel Compiler version 2021.5.0, MPLAPACK 1.0.1, MPFR 4.1.0

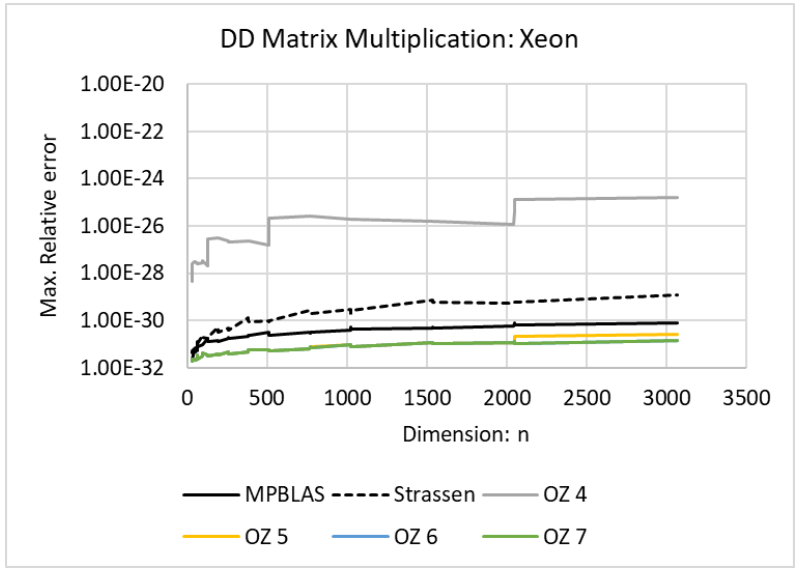
- [C++(icpc), C(icc)] -O3 -fp-model precise -qopenmp -axCORE-AVX2 -march=skylake -mtune=skylake -mcpu=skylake

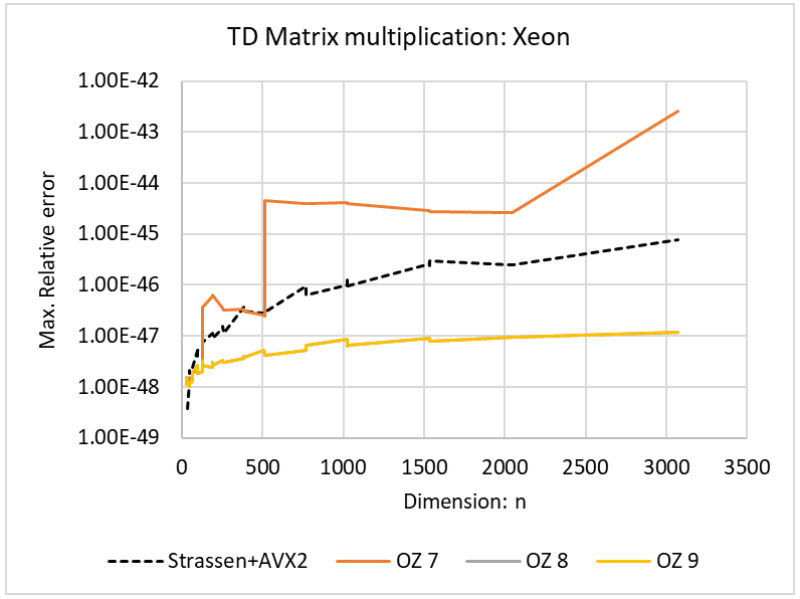
# DD, TD, QD on Xeon

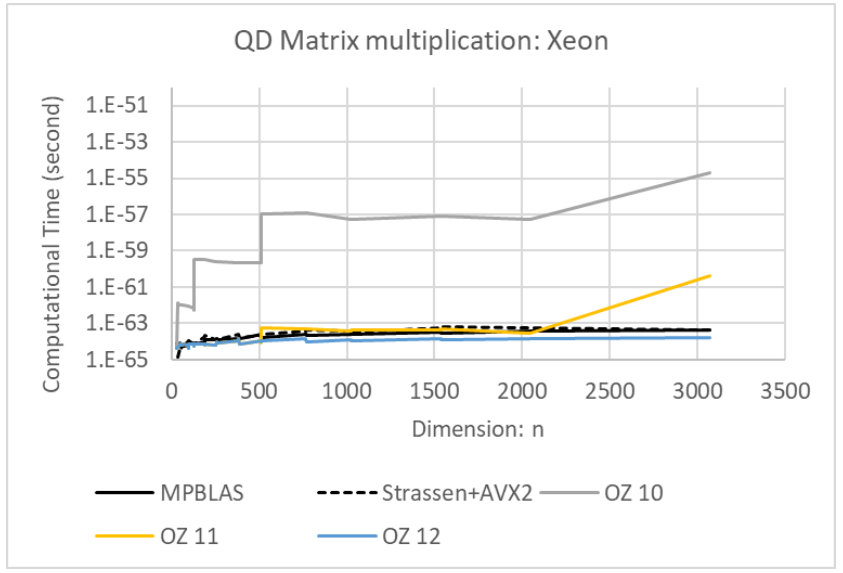


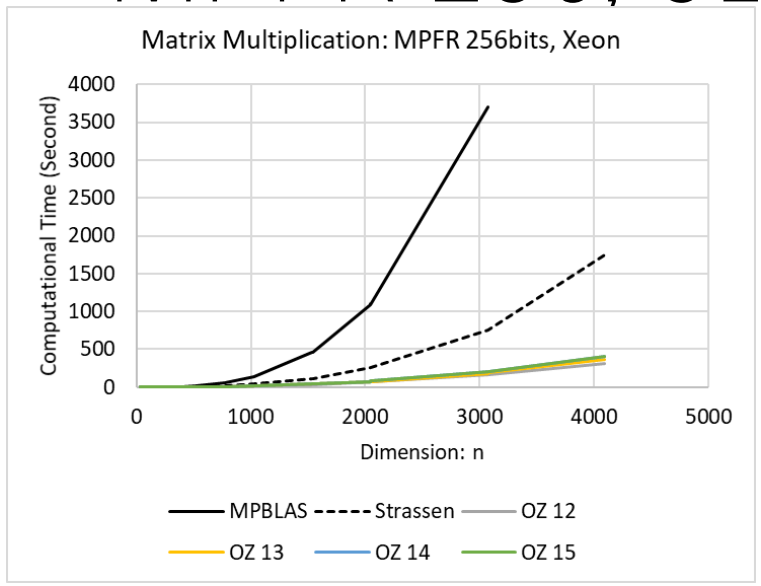



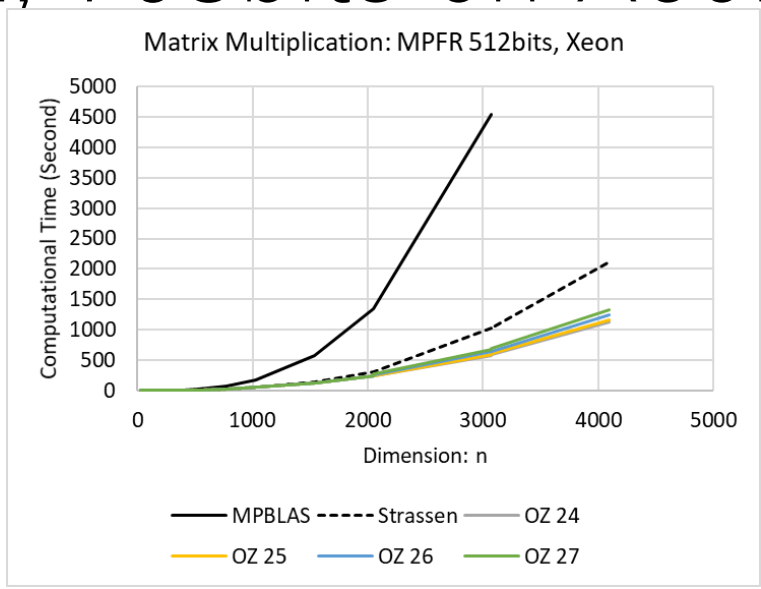



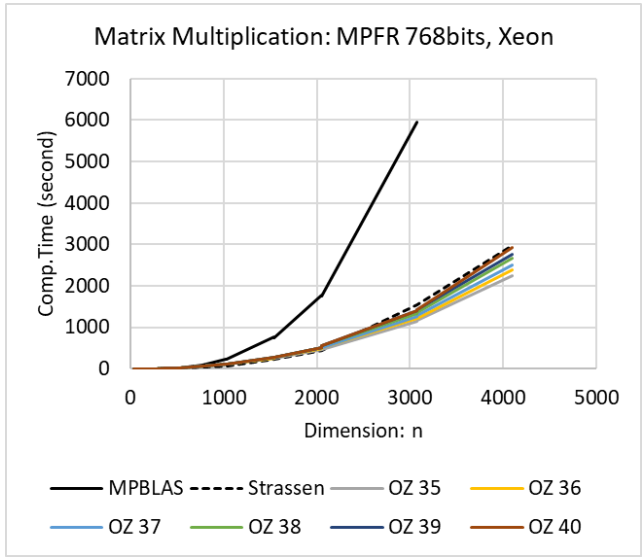



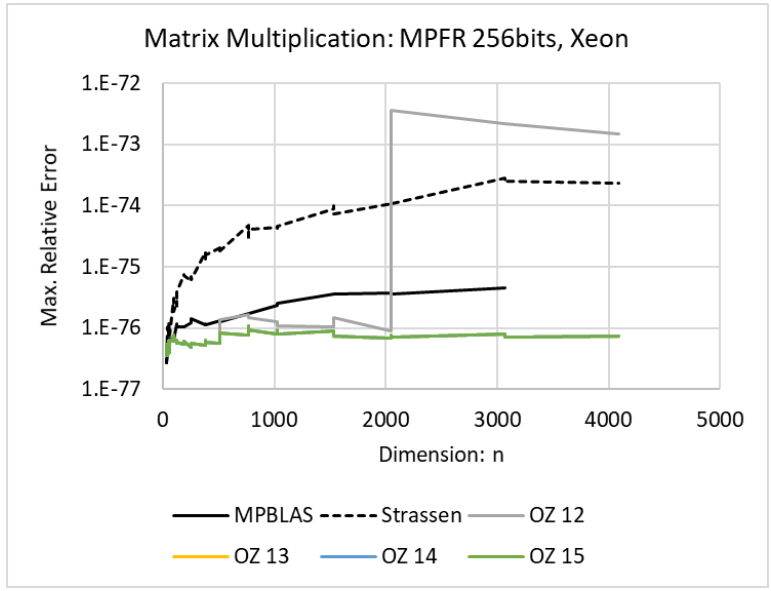



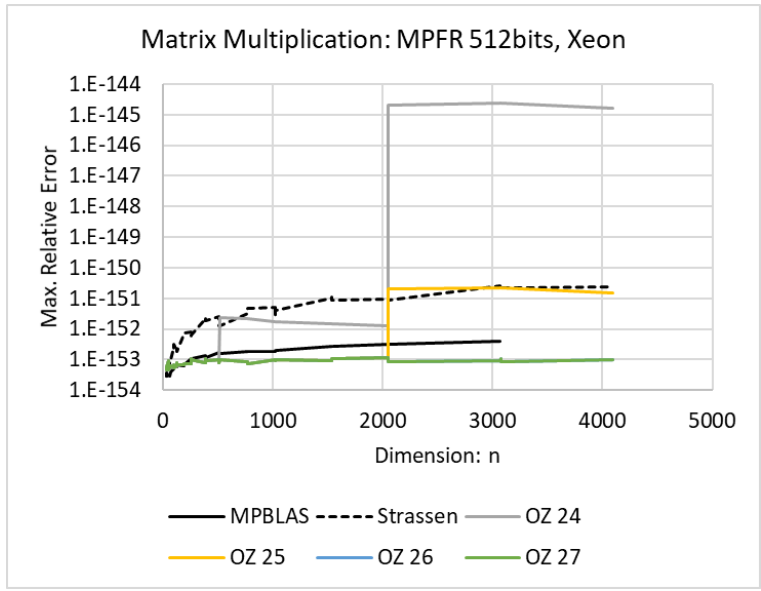
# MPFR 256, 512, 768bits on Xeon

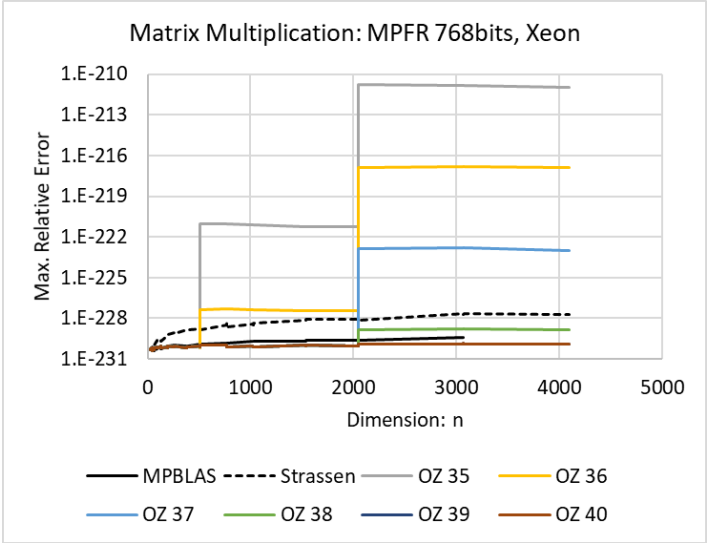


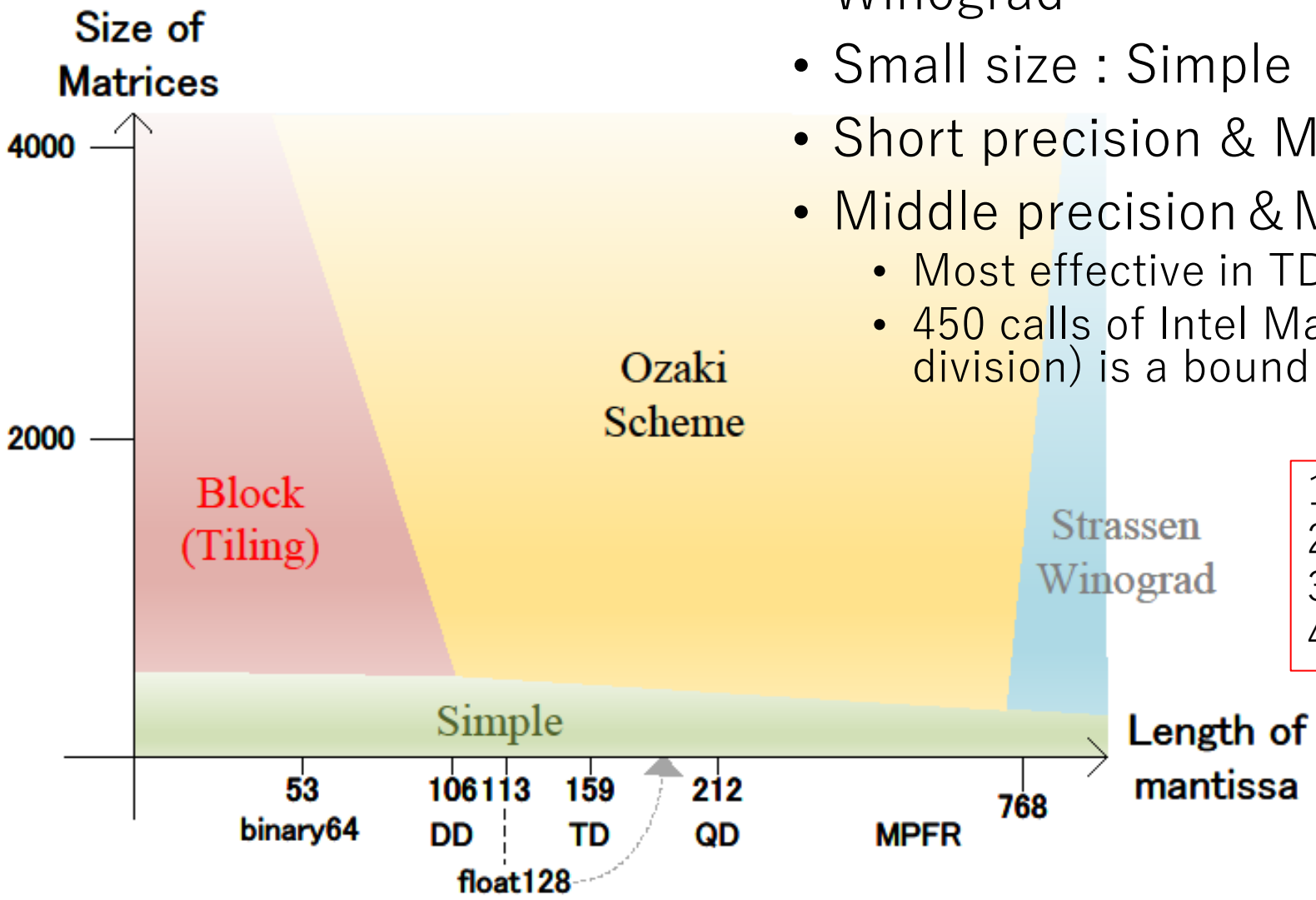



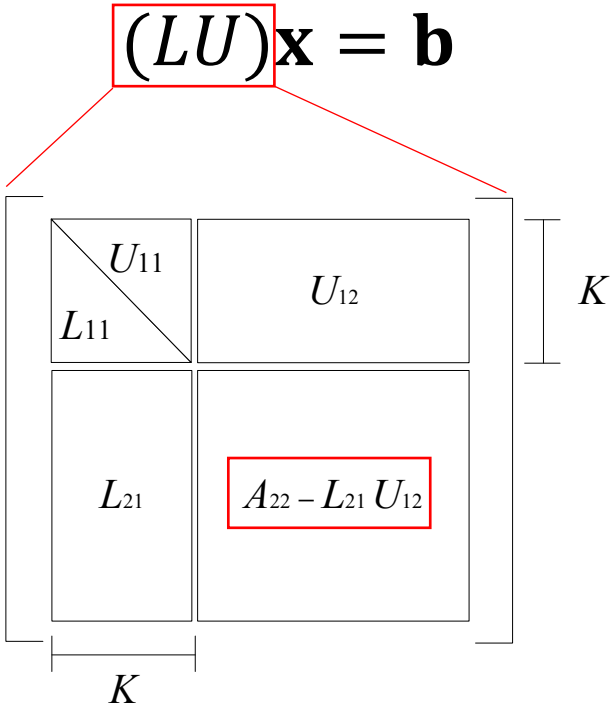
# A quick summary of optimization techniques for multiple precision matrix multiplication

- Long precision & Large size ：Strassen, Winograd
- Small size : Simple
- Short precision & Middle size: Block
- Middle precision＆Middle Size: Ozaki scheme
  - Most effective in TD – QD - MPFR 768bits precision
  - 450 calls of Intel Math Kernel GEMM (about 30 division) is a bound of effectiveness = MPFR 768bits




1. mul_[dd,td,qd,mpf]matrix_simple
2. mul_[dd,td,qd,mpf]matrix_block
3. mul_[dd,td,qd,mpf]matrix_strassen
4. mul_[dd,td,qd,mpf]matrix_oz

# Test linear system of equations




As an application of optimized matrix multiplication, we implemented LU decomposition in the direct method for various benchmark tests, including the Top500, and measured its utility on Xeon. The corresponding $n$-dimensional linear system of equation is:

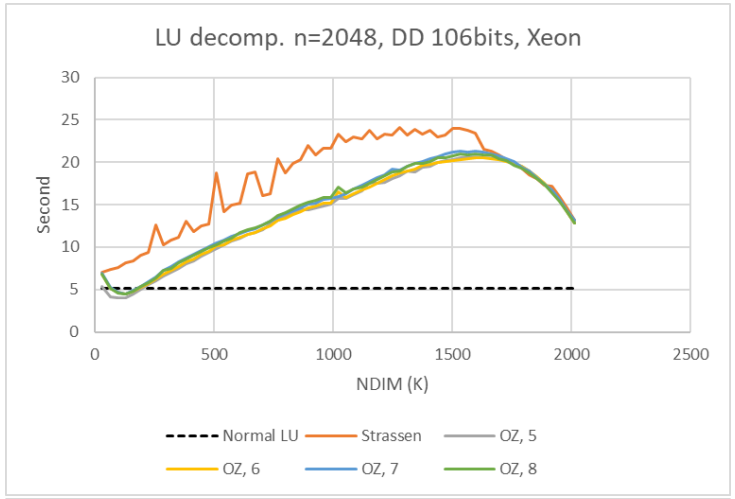
$$A\mathbf{x} = \mathbf{b}, \tag{2}$$

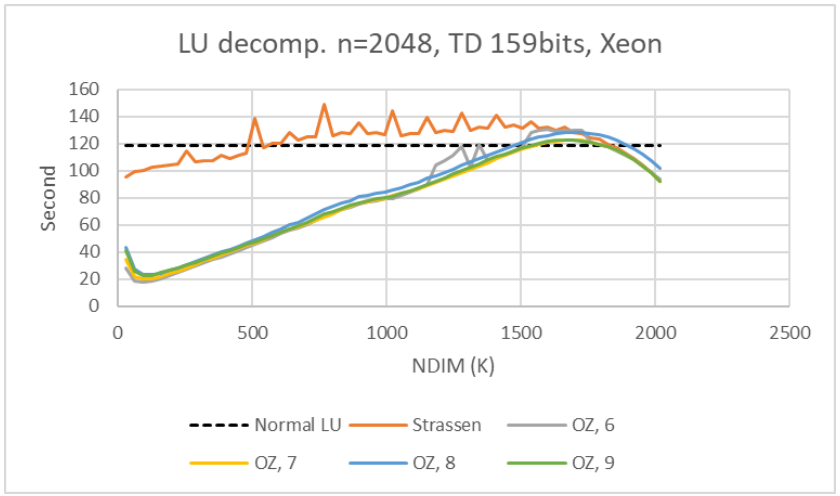
where $A \in \mathbb{R}^{n\times n}$, $\mathbf{x} \in \mathbb{R}^n$, and $\mathbf{b} \in \mathbb{R}^n$ become ill-conditional as follows:

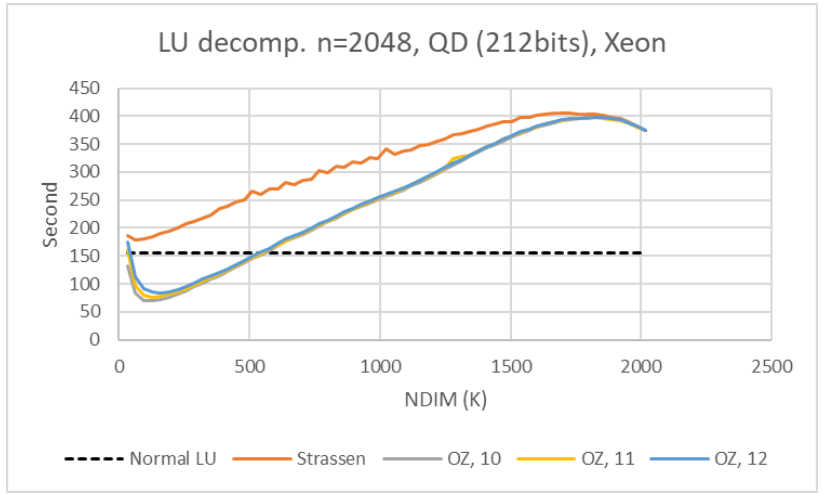
**$A$:** The diagonal matrix is $D = \mathrm{diag}[d_1 \cdots d_n]$, where $d_i := 10^{-26(i-1)/n}$. The random matrix is $R \in \mathbb{R}^{n\times n}$; then, $A$ is calculated as $A := RDR^{-1}$ using the mpmath Python library. Therefore, the condition number of $A$ is $\kappa_2(A) = \|A\|_2\|A^{-1}\|_2 = 10^{26(n-1)/n}$, which requires super-DD precision arithmetic to gain accuracy.

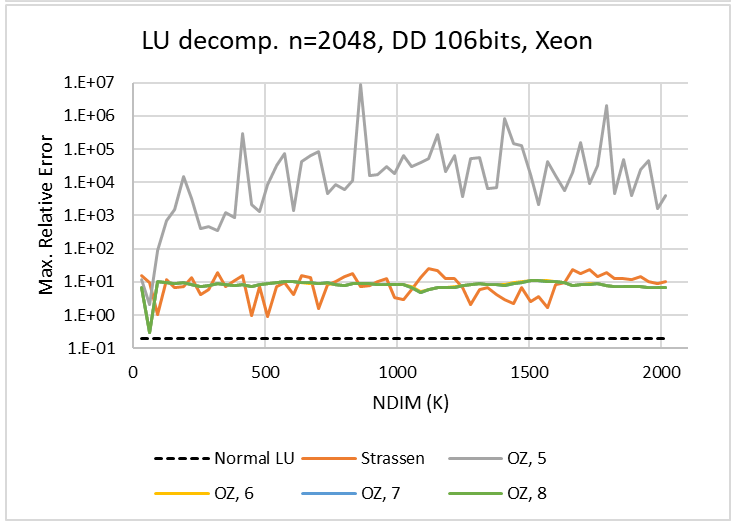
**$\mathbf{x}, \mathbf{b}$:** After setting $\mathbf{x} = [0\ 1\ \cdots\ n-1]^T$, $\mathbf{b} := A\mathbf{x}$ is calculated using mpmath.

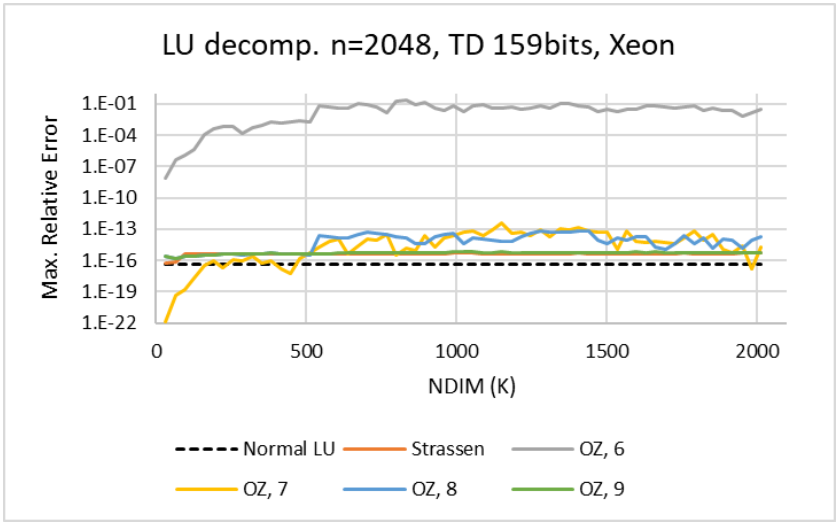
# DD, TD, QD LU on Xeon

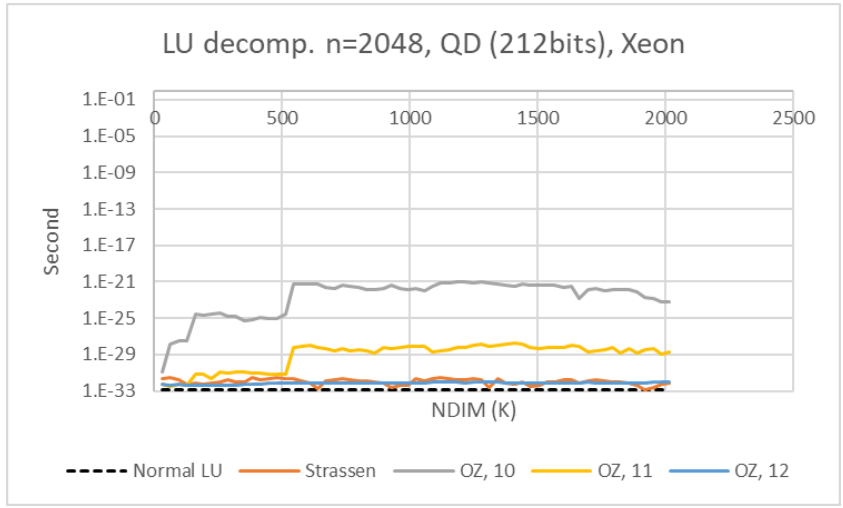


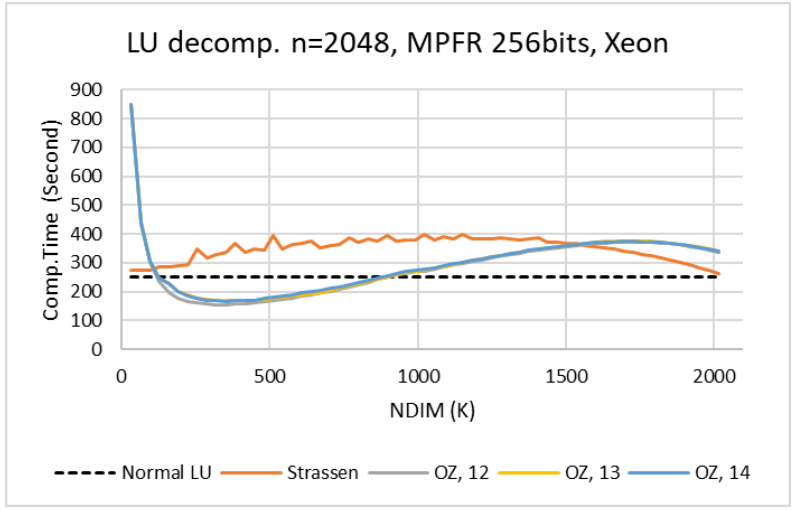



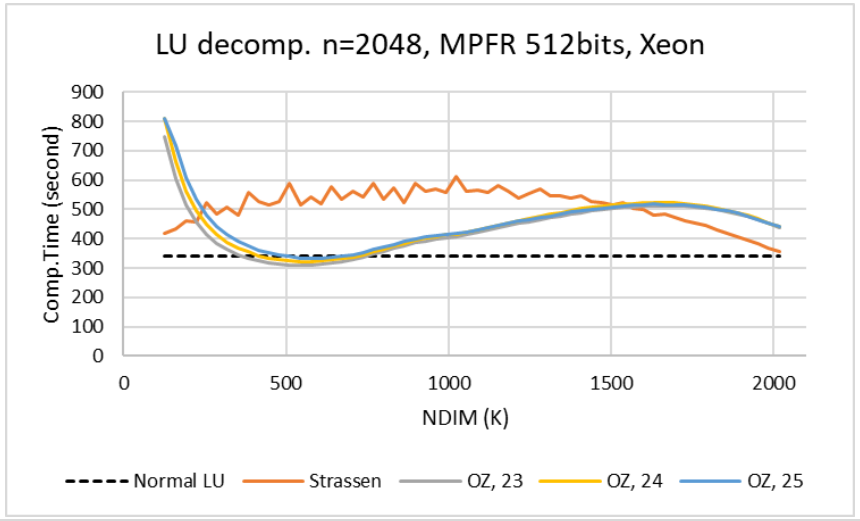



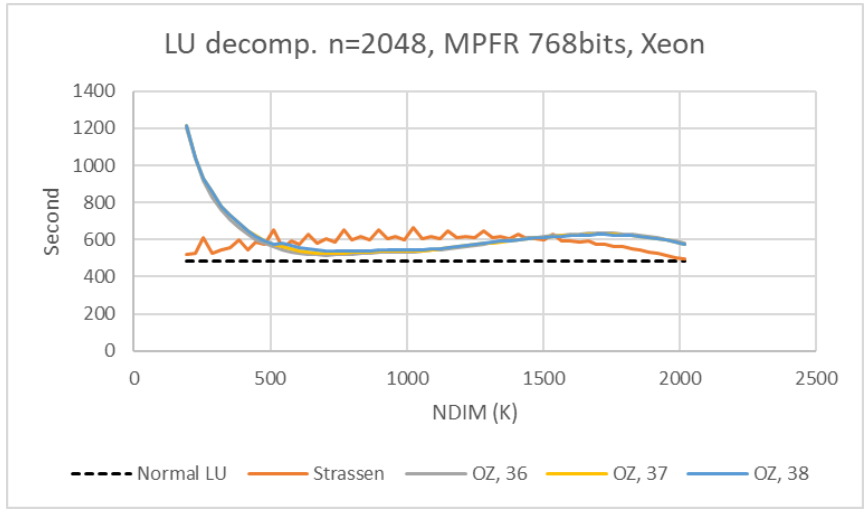



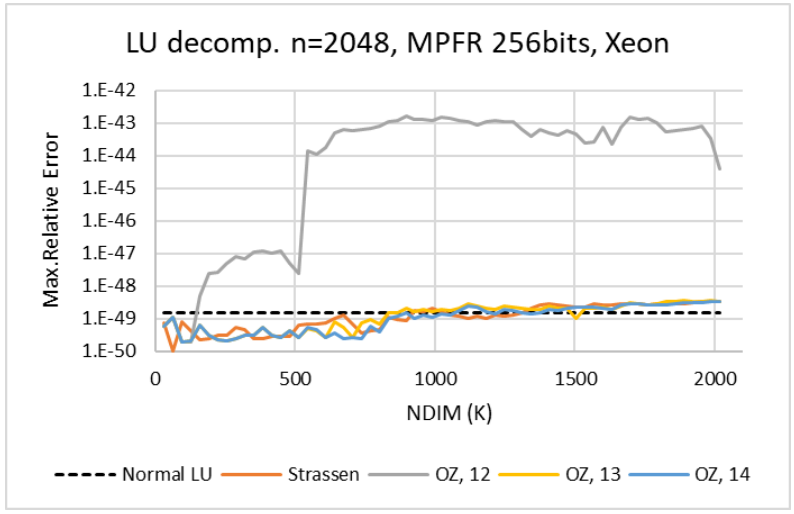

- Reduced the effectiveness due to rectangle matrix multiplication used in LU decomposition
- Trend of optimization is similar with square matrix multiplication

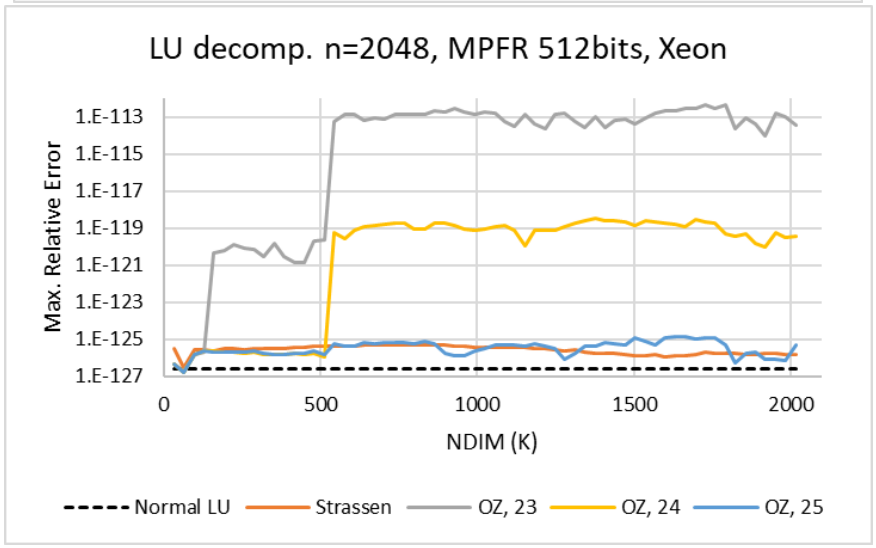
# MPFR 256, 512, 768 bits on Xeon

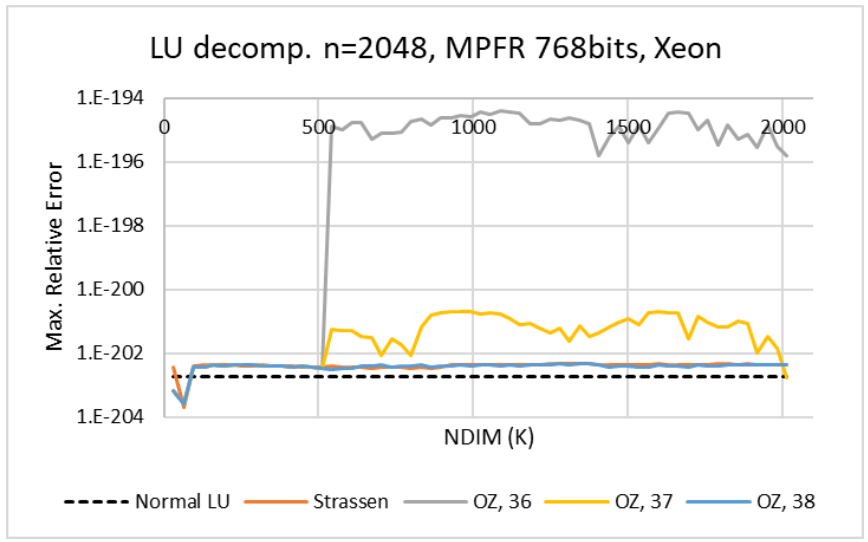



- LU w/ Ozaki scheme is not faster than row-wised LU decomposition over 768bit-precision MPFR arithmetic

# Summary: Ozaki scheme is effective for middle-precision LU decomposition

| Prec. | Method | $K$ | Second | Rel.Err. |
|---|---|---|---|---|
| DD | Rgetrf | N/A | 15.8 | 2.4E − 01 |
| 106bits | Normal LU | 1 | 5.1 | 1.9E − 01 |
| | Strassen+AVX2 | 32 | 7.4 | 1.8E + 01 |
| | OZ 6 | 128 | 4.5 | 9.4E + 01 |
| TD | Normal LU | 1 | 118.6 | 3.9E − 17 |
| 159bits | Strassen+AVX2 | 32 | 95.6 | 5.9E − 17 |
| | OZ 7 | 96 | 19.8 | 1.7E − 17 |
| QD | Rgetrf | N/A | 207.4 | 3.8E − 34 |
| 212bits | Normal LU | 1 | 155.3 | 1.4E − 33 |
| | Strassen+AVX2 | 64 | 180.8 | 1.5E − 32 |
| | OZ 12 | 160 | 83.2 | 5.8E − 33 |

| Prec. | Method | $K$ | Second | Rel.Err. |
|---|---|---|---|---|
| MPFR | Rgetrf | N/A | 398.6 | 2.1E − 50 |
| 256 bits | Normal LU | 1 | 250.0 | 1.5E − 49 |
| | Strassen | 96 | 273.7 | 8.3E − 50 |
| | OZ 13 | 320 | 167.6 | 3.2E − 50 |
| MPFR | Rgetrf | N/A | 492.3 | 2.5E − 127 |
| 512 bits | Normal LU | 1 | 341.1 | 2.6E − 127 |
| | Strassen | 32 | 390.4 | 3.1E − 126 |
| | OZ 25 | 576 | 333.4 | 4.1E − 126 |
| MPFR | Rgetrf | N/A | 627.8 | 5.1E − 205 |
| 768 bits | Normal LU | 1 | 481.4 | 1.9E − 203 |
| | Strassen | 32 | 491.1 | 3.6E − 203 |
| | OZ 38 | 736 | 536.6 | 3.7E − 203 |

# Complex GEMM: 4M and 3M methods

- Multiplication of Complex matrices

$$A \in \mathbb{C}^{m\times l}, B \in \mathbb{C}^{l\times n} \Rightarrow AB \in \mathbb{C}^{m\times n}$$

- 4M method：4 real multiplications, 2 real additions

$$AB = \{\mathrm{Re}(A)\mathrm{Re}(B) - \mathrm{Im}(A)\mathrm{Im}(B)\} + \{\mathrm{Re}(A)\mathrm{Im}(B) - \mathrm{Im}(A)\mathrm{Re}(B)\}\cdot \mathrm{i} \in \mathbb{C}^{m\times n}$$

- 3M method：3 real multiplications, 5 real additions

$$T_1 = \mathrm{Re}(A)\mathrm{Re}(B) \in \mathbb{R}^{m\times n},\ T_2 = \mathrm{Im}(a)\mathrm{Im}(b) \in \mathbb{R}^{m\times n}$$

$$AB = (T_1 - T_2) + \{\big(\mathrm{Re}(A) + \mathrm{Im}(A)\big)\big(\mathrm{Re}(B) + \mathrm{Im}(B)\big) - T_1 - T_2\}\cdot \mathrm{i} \in \mathbb{C}^{m\times n}$$

**Implementation of 3M method**

1. MPC library https://www.multiprecision.org/mpc/
2. Bini: MPsolver GNU MP's mpf-based 3M method
3. BLIS: 3M and 4M CGEMM based on DGEMM

# Benchmark test of CGEMM(1/3)

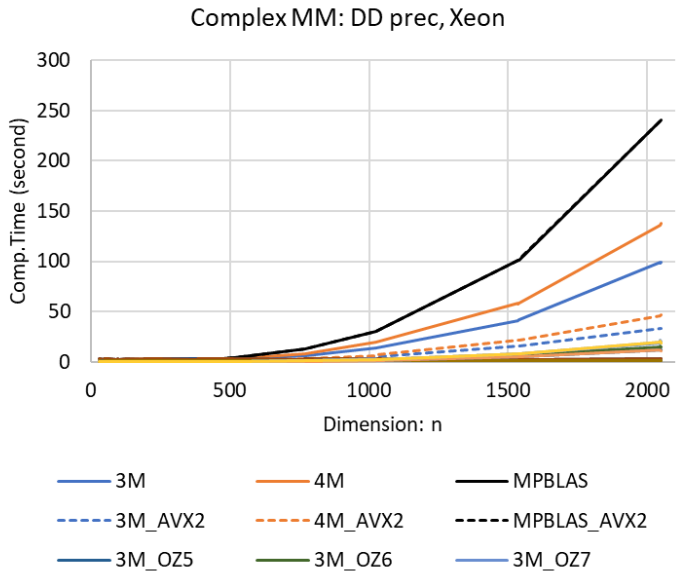
$$A,\ B \in \mathbb{C}^{n \times n} \quad \Longrightarrow \quad C := AB$$

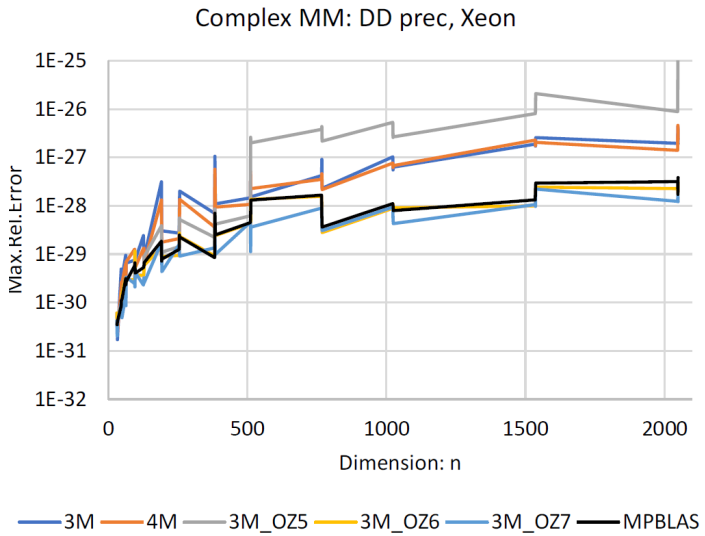
Real and imaginary parts of each element

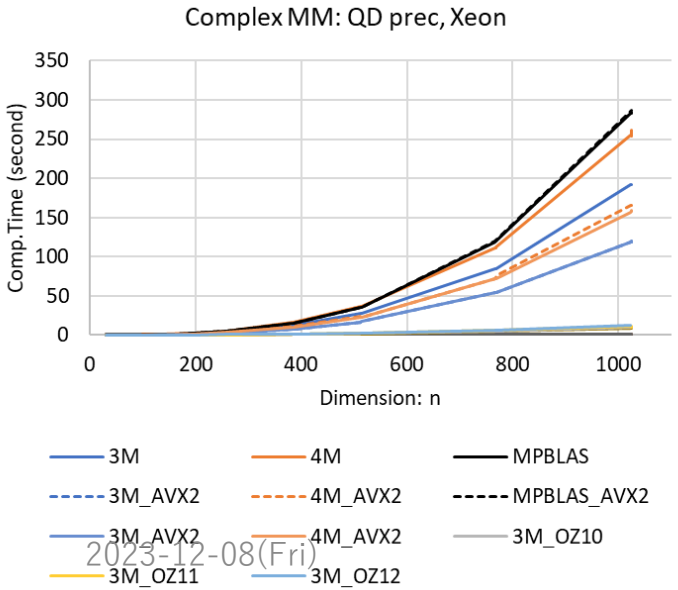
$$(ru - 0.5) \times \exp(rn)$$

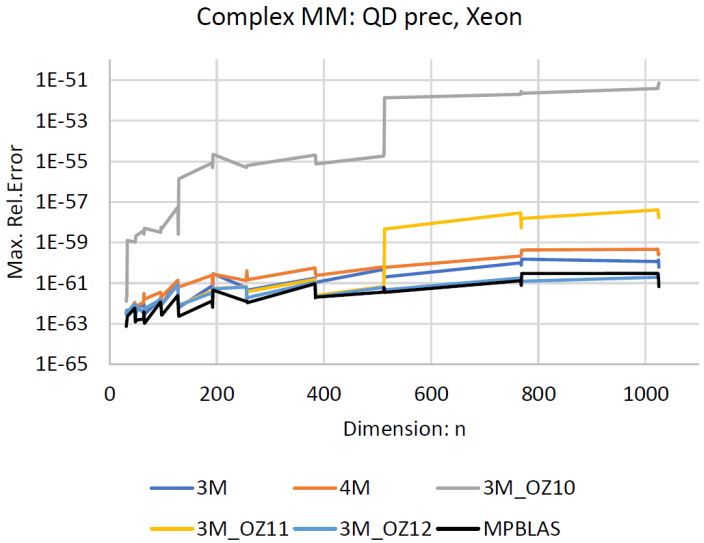
[CPU and OS] Intel Xeon W-2295 3.0GHz 18 cores, Ubuntu 20.04.3 LTS
[C/C++] Intel Compiler version 2021.5.0
[Compiler Option] -O3 -std=c++11 -fp-model precise
[with AVX2] -axCORE-AVX2 -march=skylake -mtune=skylake mcpu=skylake
[MPLAPACK] 2.0.1, GNU MP 6.2.1, MPFR 4.1.0,  MPC 1.2.1

# Benchmark test of CGEMM(2/3): DD and QD prec.



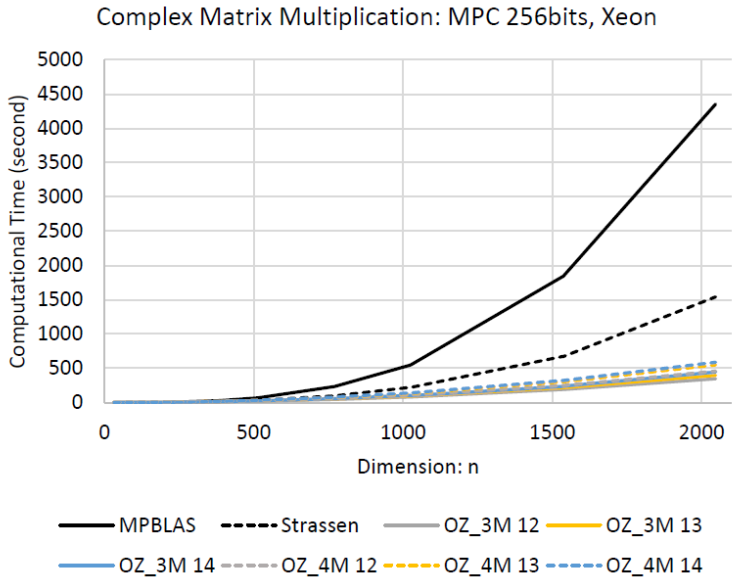



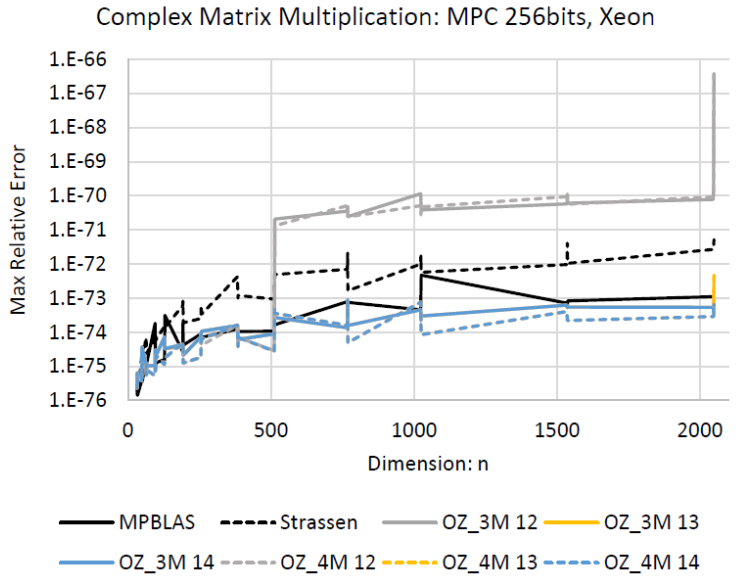



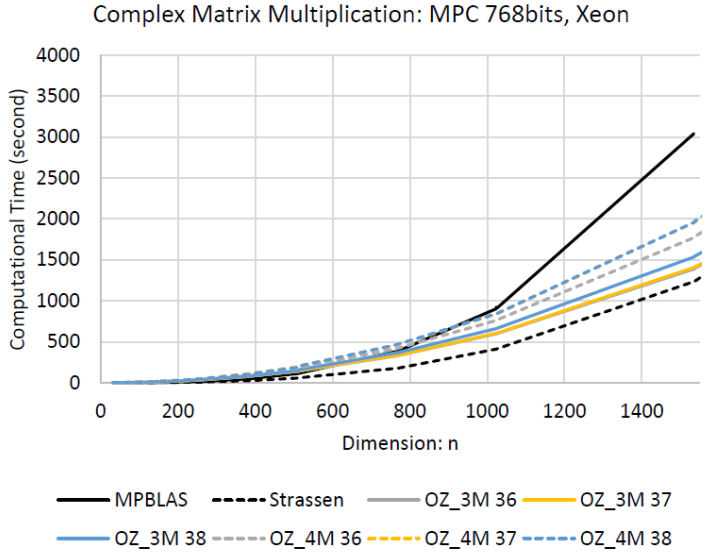



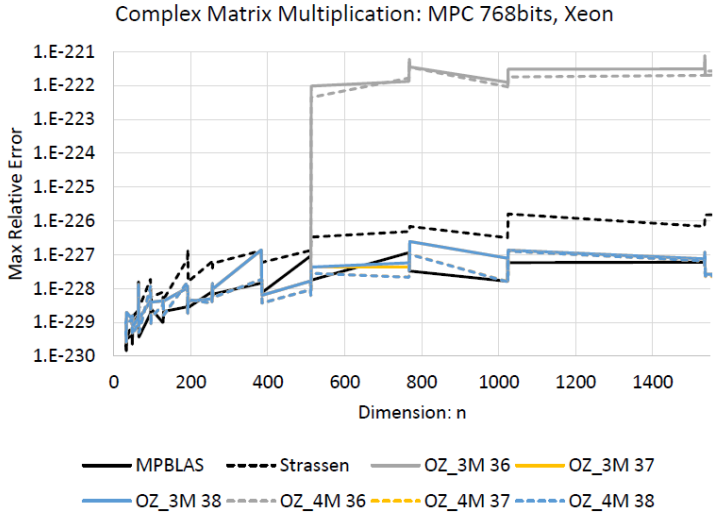

- Ozaki scheme(DD, QD + IMKL DGEMM)
  - DD prec.→6 divisions
  - QD prec.→12 divisions
- There is no difference of accuracy between 4M and 3M methods

# Benchmark test of CGEMM(3/3): 256bits and 768bits



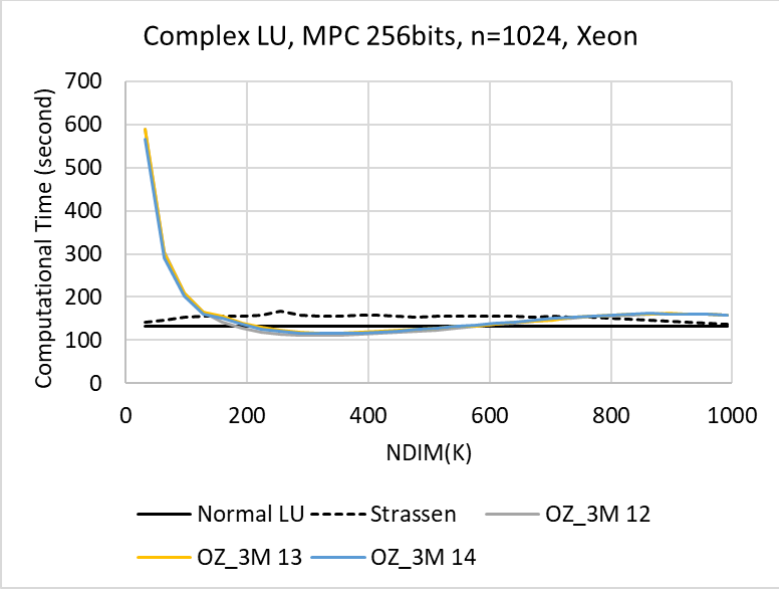



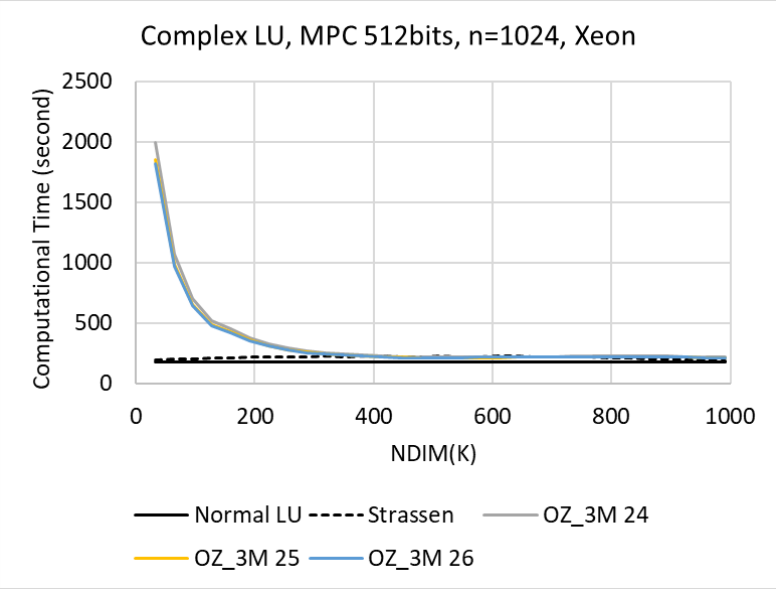



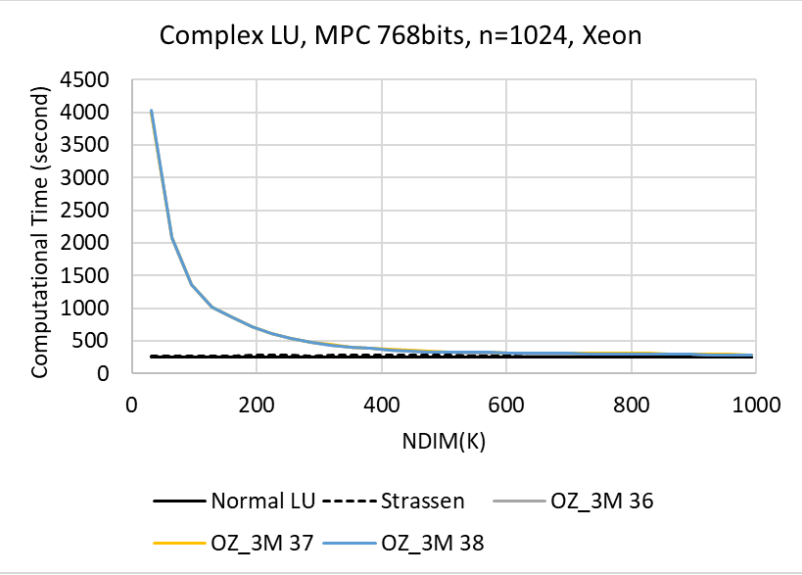



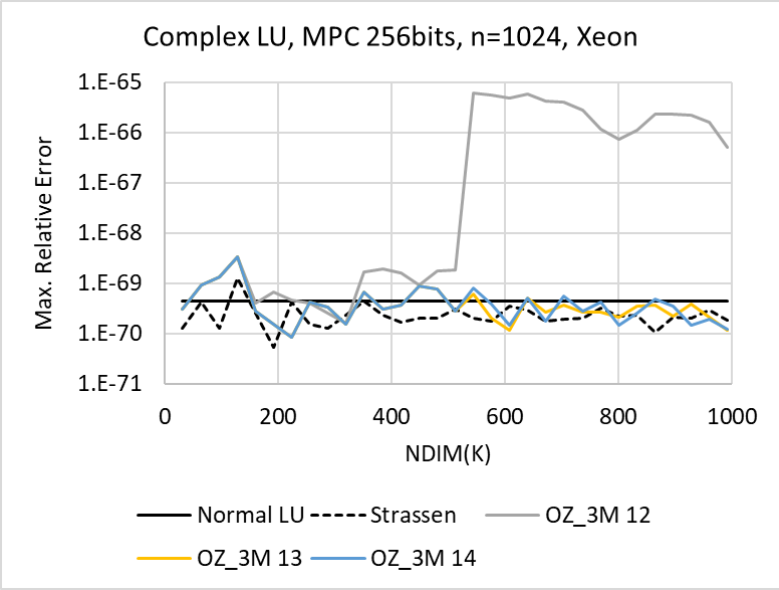

- MPBLAS and Strassen are built on MPC
- 3M Ozaki scheme is made of IMKL DGEMM
- 256bits→13 divisions →Ozaki scheme is the fastest
- 768bits→37 divisions→Strassen is the fastest
- The same result as on real GEMM benchmark

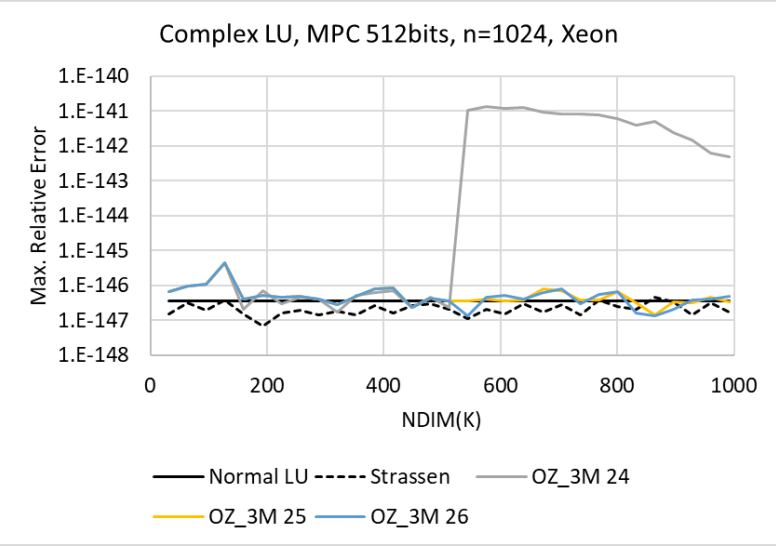
# Complex LU : MPC 256, 512, 768 bits on Xeon

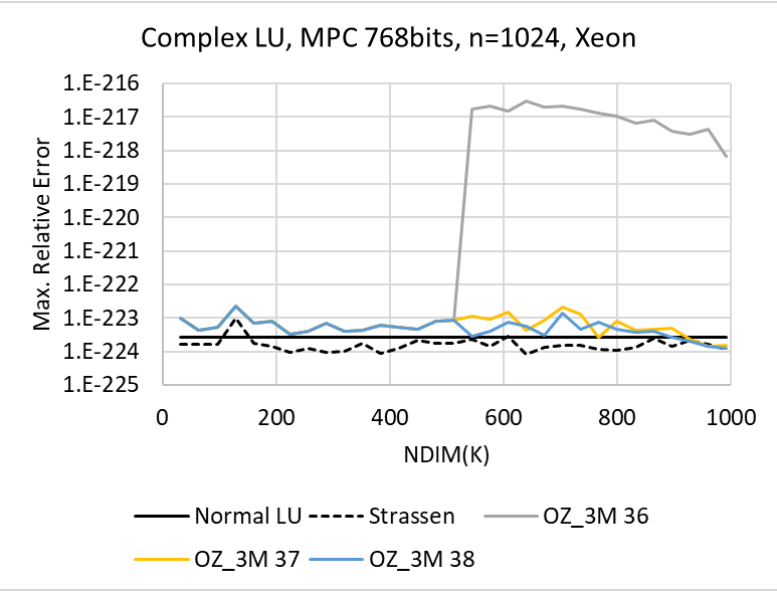


# Summary: Ozaki scheme is effective for middle-precision LU decomposition

TABLE II: Minimum computational time (s) of MPC LU decomposition and maximum relative errors of $\mathbf{x}$ on Xeon

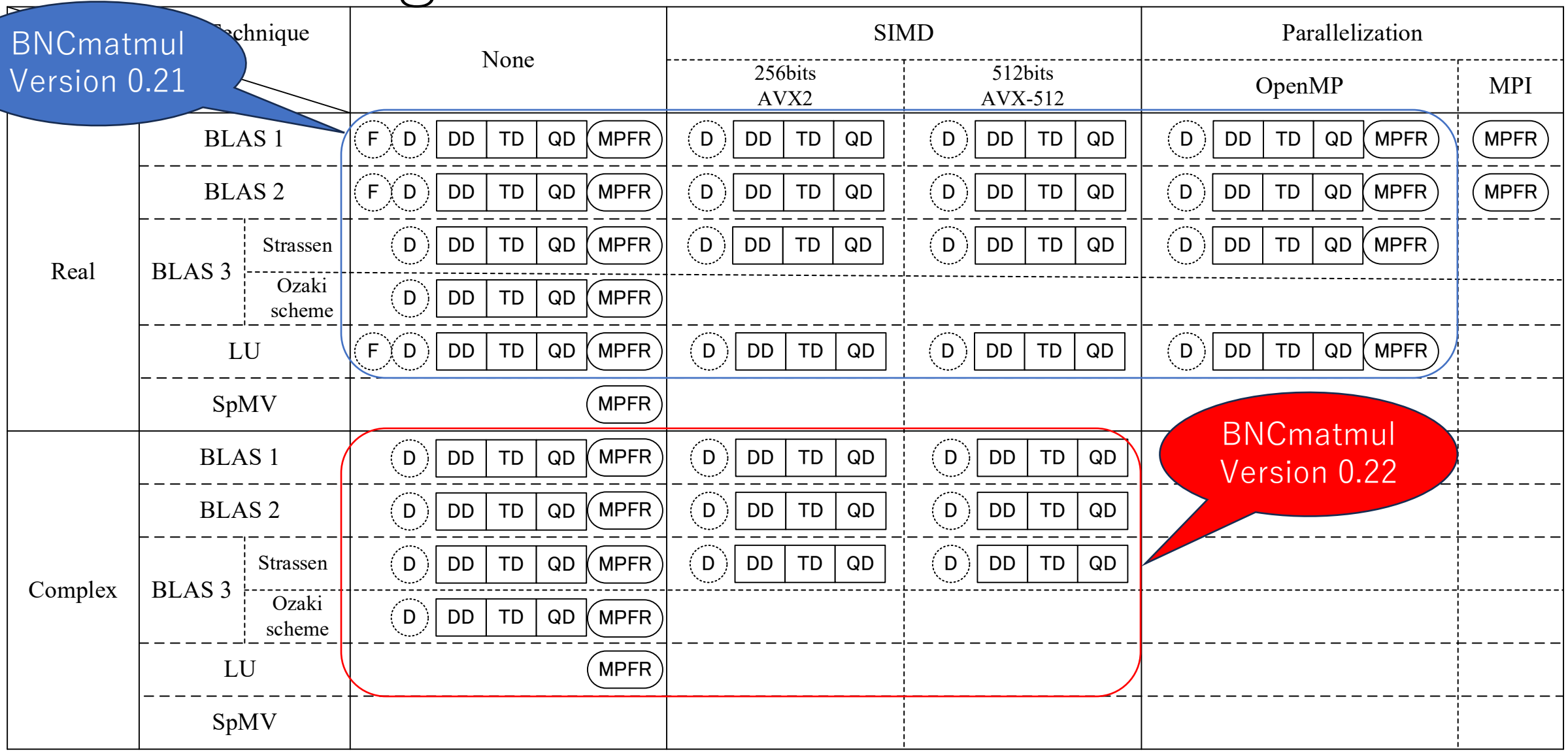
| Prec. | Method | $K$ | Second | Rel.Err. |
|---|---|---|---|---|
| MPC 256 bits | Cgetrf | N/A | 188.1 | 2.9E − 71 |
| | Normal LU | 1 | 134.1 | 4.5E − 70 |
| | Strassen | 992 | 136.1 | 1.9E − 70 |
| | OZ 13 | 352 | 116.6 | 6.9E − 70 |
| MPC 512 bits | Cgetrf | N/A | 236.7 | 6.4E − 149 |
| | Normal LU | 1 | 181.7 | 3.7E − 147 |
| | Strassen | 992 | 187.6 | 8.0E − 147 |
| | OZ 25 | 992 | 209.0 | 3.4E − 147 |
| MPC 768 bits | Cgetrf | N/A | 313.7 | 1.6E − 225 |
| | Normal LU | 1 | 253.0 | 2.6E − 224 |
| | Strassen | 992 | 257.5 | 1.1E − 224 |
| | OZ 37 | 992 | 287.6 | 1.5E − 224 |

TABLE III: Minimum computational time (s) of MPC LU decomposition and maximum relative errors of $\mathbf{x}$ on EPYC

| Prec. | Method | $K$ | Second | Rel.Err. |
|---|---|---|---|---|
| MPC 256 bits | Cgetrf | N/A | 275.9 | 2.9E − 71 |
| | Normal LU | 1 | 184.1 | 4.5E − 70 |
| | Strassen | 32 | 182.8 | 1.3E − 70 |
| | OZ 13 | 384 | 171.1 | 3.1E − 70 |
| MPC 512 bits | Cgetrf | N/A | 327.7 | 6.4E − 149 |
| | Normal LU | 1 | 237.8 | 3.7E − 147 |
| | Strassen | 64 | 240.7 | 3.2E − 147 |
| | OZ 25 | 992 | 279.6 | 3.4E − 147 |
| MPC 768 bits | Cgetrf | N/A | 428.4 | 1.6E − 225 |
| | Normal LU | 1 | 334.9 | 2.6E − 224 |
| | Strassen | 32 | 333.5 | 1.6E − 224 |
| | OZ 37 | 992 | 377.5 | 1.5E − 224 |

- The computational time is slower than on the Xeon: about 20% slower for MPC alone, and up to 40¥% slower for the Ozaki scheme with DGEMM.
- There is little difference in the maximum relative error.

# List of categorized functions in BNCmatmul



# III. 応用事例

# A) 混合精度反復改良法

# 直接法ベースの混合精度反復改良法

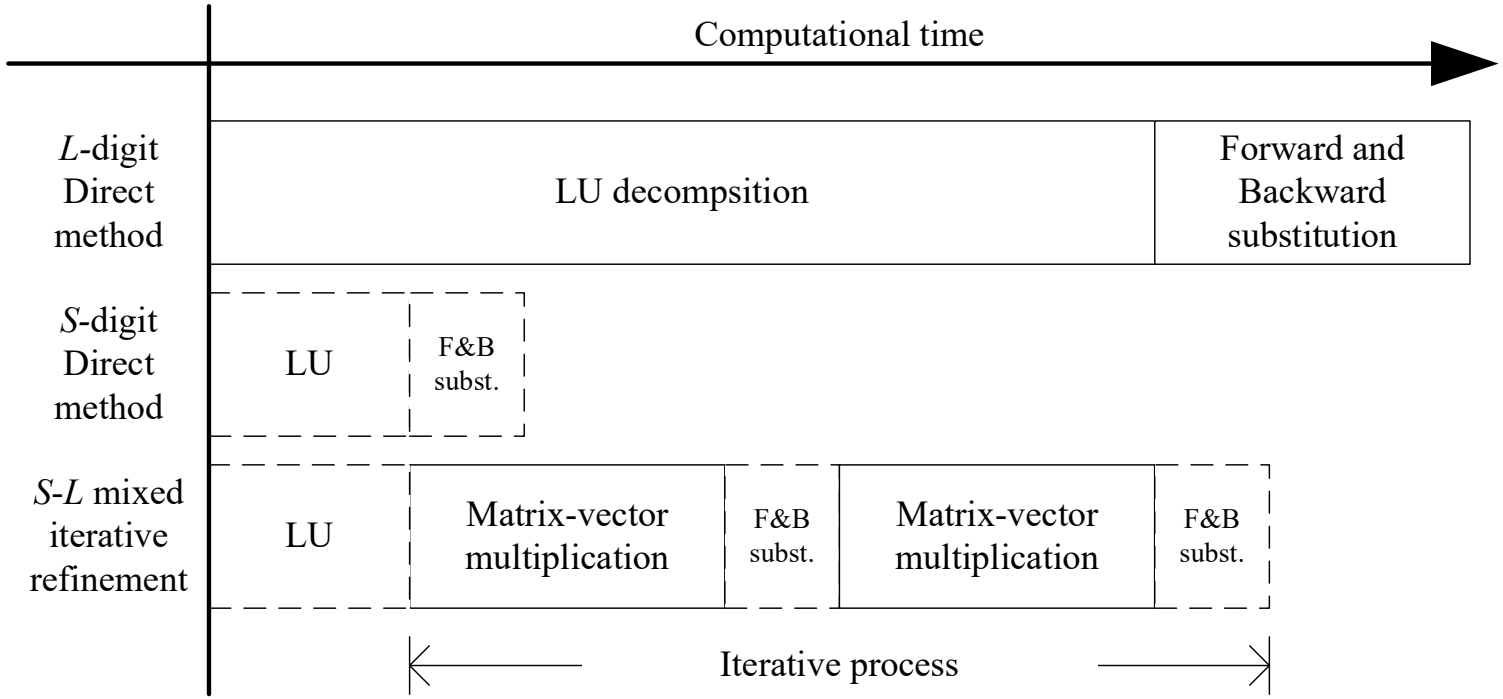
Mixed precision iterative refinement method is to reduce computational cost by combining short $S$ digits arithmetic and long $L$ digits arithmetic ($S << L$).

(cf.) Buttari, Alfredo, et al. International Journal of High Performance Computing Applications 21.4 (2007): 457-466

連立一次方程式: $A\,\mathbf{x} = \mathbf{b},\ A \in \mathbb{R}^{n\times n},\ \mathbf{b}, \mathbf{x} \in \mathbb{R}^{n}$

1. $(S)$ Solve $A\,\mathbf{x_0} = \mathbf{b}$ for $\mathbf{x_0}$
2. For $\nu = 0, 1, 2, \ldots$
3. $\quad (L)\ \mathbf{r}_\nu := \mathbf{b} - A\mathbf{x}_\nu$
4. $\quad (L \to S)\ \mathbf{r}'_\nu := \mathbf{r}_\nu \,/\, \|\mathbf{r}_\nu\|$
5. $\quad (S)$ Solve $A\,\mathbf{z} = \mathbf{r}_\nu{}'$ for $\mathbf{z}$.
6. $\quad (L)\ \mathbf{x}_{\nu+1} := \mathbf{x}_\nu + \|\mathbf{r}_\nu\|\,\mathbf{z}$
7. $\quad (L)$ Check convergence.

→ $\mathbf{x} := \mathbf{x}_{\nu_{\mathrm{stop}}}$



# 混合精度反復改良法のPythonコード

```python
def iterative_refinement_dd_mp(a, b, maxtimes, rtol, atol):
    dim = b.rows

    # Initialize
    af_ch = (ct.c_long * (dim))()
    af = init_ddmatrix(dim, dim)
    bf = init_ddvector(dim)
    xf = init_ddvector(dim)

    x = init_mpvector(dim)
    res = init_mpvector(dim)

    resf = init_ddvector(dim)
    z = init_mpvector(dim)
    zf = init_ddvector(dim)

    # norm_a := ||A||_F
    norm_a = mpmath.mnorm(a, 'fro')

    # Make short precision copy of A and b
    af = get_ddmatrix_mpmat(af, a)
    bf = get_ddvector_mpvec(bf, b)

    # Compute LU factorization in short precision
    DDLUdecompPM(af, af_ch)
    SolveDDLSPM(xf, af, bf, af_ch)

    # Promote te solution from short precision to long precision
    x = set_ddvector_mpvec(x, xf)
```

```python
    # repeat iterative refinement process
    for itimes in range(maxtimes):
        # Compute residual in long precision
        res = b - a * x

        # until ||r_i||_2 < sqrt(n) * reps * ||A||_F * ||x_i||_2
        norm_x = mpmath.norm(x); norm_res = mpmath.norm(res)
        if norm_res < mpmath.sqrt(mpmath.mpf(dim)) * rtol * norm_a * norm_x +
        atol:
            break

        # normalization: res := coef * res
        normalization_coef = mpmath.mpmathify(1) / norm_res
        res = normalization_coef * res

        # Demote the residual from long precision to short precison
        resf = get_ddvector_mpvec(resf, res)

        # Back-solve on short precision residual and short precision factors
        SolveDDLSPM(zf, af, resf, af_ch)

        # Promote the correction from short precision to long precision
        z = set_ddvector_mpvec(z, zf)

        # reverse normalizationi
        z = norm_res * z

        # Update solution in long precision
        x = x + z
```

# ベンチマークテスト: 424bits, rtol=$10^{-100}$

$A$ We produce $A := RDR^{-1}$ with the appropriate precision mpmath, where the real diagonal matrix $D = \mathrm{diag}[d_1 \cdots d_n]$ ($d_i := 10^{-26(i-1)/n}$) and real random matrix $R \in \mathbb{R}^{n\times n}$. Therefore, $\kappa_2(A) = \|A\|_2\|A^{-1}\|_2 = 10^{26(n-1)/n}$.

$\mathbf{x}, \mathbf{b}$ The true solution is $\mathbf{x} = [0\ 1\ \cdots\ n-1]^T$, and we calculate $\mathbf{b} := A\mathbf{x}$ with mpmath.

| | Corei9: $n = 2000$ | | | EPYC: $n = 2000$ | | |
|---|---|---|---|---|---|---|
| Prec. iter. refinement | Time(s) | Max.RE | #Iter. | Time(s) | Max.RE | #Iter. |
| DD+MP (424 bits) | 190.7 | 2.2E-78 | 15 | 294.2 | 2.2E-78 | 16 |
| DD+MP (AVX2) | 177.0 | 2.7E-77 | 15 | 279.4 | 2.7E-77 | 15 |
| TD+MP | 171.0 | 1.0E-80 | 2 | 241.0 | 1.0E-80 | 2 |
| TD+MP (AVX2) | 131.3 | 8.8E-80 | 2 | 200.0 | 8.8E-80 | 2 |
| QD+MP | 409.8 | 7.5E-99 | 1 | 564.6 | 7.5E-99 | 1 |
| QD+MP (AVX2) | 176.5 | 1.8E-98 | 1 | 287.9 | 1.8E-98 | 1 |
| MP direct (424 bits) | 17940.8 | 8.8E-99 | N/A | 27700.3 | 8.8E-99 | N/A |

- AVX2による高速化の効果は DD+MPよりTD+MP, QD+MPで顕著(反復回数に依存)
- 最小の計算時間はTD+MPとなった

# B) 陰的Runge-Kutta法の高速化（昔の話題）

# IVP of $n$-th dimensional ODE to be solved

$$\begin{cases} \frac{d\mathbf{y}}{dt} = \mathbf{f}(t, \mathbf{y}) \ \in \mathbb{R}^n \\ \mathbf{y}(t_0) = \mathbf{y}_0 \end{cases} \tag{1}$$

Integration Interval:$[t_0, \alpha]$

We suppose that this above ODE has the unique solution, so Lipschize constant $L > 0$ exists to be satisfied such as

$$||\mathbf{f}(t, \mathbf{v}) - \mathbf{f}(t, \mathbf{w})|| \leq L||\mathbf{v} - \mathbf{w}|| \tag{2}$$

for $\forall \mathbf{v}, \mathbf{w} \in \mathbb{R}^n$, $\forall t \in [t_0, \alpha]$.

$$\Downarrow$$

1D Brusselator problem and K-S eq. has large $L >> 1$, so they are called "stiff problems. "

# Skeleton of $m$ stages IRK methods

Discretization : $t_0,\ t_1 := t_0 + h_0,\ \ldots,\ t_{k+1} := t_k + h_k \ldots$

When we calculate the approximation $\mathbf{y}_{k+1} \approx \mathbf{y}(t_{k+1})$ from the former $\mathbf{y}_k \approx \mathbf{y}(t_k)$, the following two steps are executed:

(A) Inner iteration: Solve the nonlinear equation for unknown $\mathbf{Y} = [Y_1 \ \ldots \ Y_m]^T \in \mathbb{R}^{mn}$.

$$\begin{cases} Y_1 &= \mathbf{y}_k + h_k \sum_{j=1}^m a_{1j} \mathbf{f}(t_k + c_j h_k, Y_j) \\ &\vdots \\ Y_m &= \mathbf{y}_k + h_k \sum_{j=1}^m a_{mj} \mathbf{f}(t_k + c_j h_k, Y_j) \end{cases}$$

$$\Updownarrow$$

$$\mathbf{F}(\mathbf{Y}) = 0 \tag{3}$$

(B) Calculate the next approximation $\mathbf{y}_{k+1}$ with the above $\mathbf{Y}$.

$$\mathbf{y}_{k+1} := \mathbf{y}_k + h_k \sum_{j=1}^m b_j \mathbf{f}(t_k + c_j h_k, Y_j)$$

# The whole algorithm of accelerated IRK method

Initial guess: $\mathbf{Y}_{-1} \in \mathbb{R}^{mn}$

For $l = 0, 1, 2, ...$ Simplified Newton iteration

(1) $\mathbf{Y}_l := [Y_1^{(l)}\ Y_2^{(l)}\ ...\ Y_m^{(l)}]^T$

(2) $C := I_m \otimes I_n - h_k X \otimes J$, Compute $||C||_F$

(3) $\mathbf{d} := (W^T B \otimes I_n)(-\mathbf{F}(\mathbf{Y}_l))$

(4) Solve $C\mathbf{x}_0 = \mathbf{d}$ for $\mathbf{x}_0$ $(S)$

For $\nu = 0, 1, 2, ...$ Mixed precision iterative refinement

(5) $\mathbf{r}_\nu := \mathbf{d} - C\mathbf{x}_\nu$

(6-1) $\mathbf{r}'_\nu := \mathbf{r}_\nu / ||\mathbf{r}_\nu||$ $(S)$

(6-2) Solve $C\mathbf{z} = \mathbf{r}'_\nu$ for $\mathbf{z}$ $(S)$

(6-3) $\mathbf{x}_{\nu+1} := \mathbf{x}_\nu + ||\mathbf{r}_\nu||\mathbf{z}$

(6-4) Check convergence $\Rightarrow \mathbf{x}_{\nu_{stop}}$

(7) $\mathbf{Y}_{l+1} := \mathbf{Y}_l + (W \otimes I_n)\mathbf{x}_{\nu_{stop}}$

Check convergence $\Rightarrow \mathbf{Y}_{l_{stop}}$

$\mathbf{Y} := \mathbf{Y}_{l_{stop}} = [Y_1\ Y_2\ ...\ Y_m]^T$

$\mathbf{y}_{k+1} := \mathbf{y}_k + h_k \sum_{j=1}^{m} b_j \mathbf{f}(t_k + c_j h_k, Y_j)$

# 1D Brusselator Problem

$$\begin{cases} \frac{\partial u}{\partial t} = 1 + u^2 v - 4 + 0.02 \cdot \frac{\partial^2 u}{\partial x^2} \\ \frac{\partial v}{\partial t} = 3u - u^2 v + 0.02 \cdot \frac{\partial^2 v}{\partial x^2} \end{cases}$$

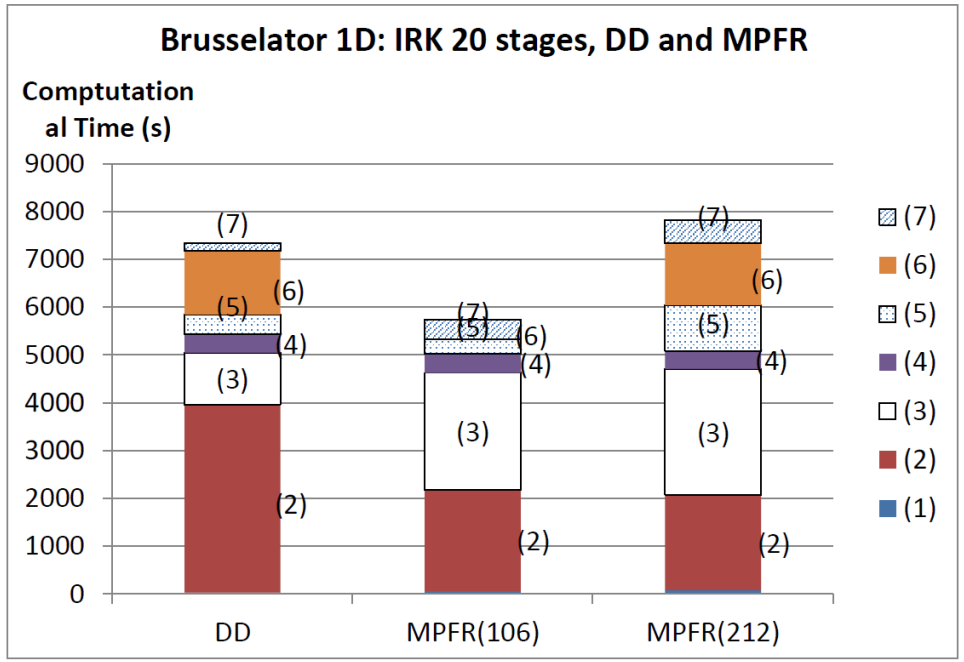
$$\Downarrow$$

$$\begin{cases} \frac{du_i}{dt} = 1 + u_i^2 v_i - 4 + 0.02 \cdot \frac{u_{i+1} - 2u_i + u_{i-1}}{(\Delta x)^2} \\ \frac{dv_i}{dt} = 3u_i - u_i^2 v_i + 0.02 \cdot \frac{v_{i+1} - 2v_i + v_{i-1}}{(\Delta x)^2} \\ \Delta x = 1/(N+1) \\ u_0(t) = u_{N+1}(t) = 1,\ v_0(t) = v_{N+1}(t) = 3, \\ u_i(0) = 1 + \sin(2\pi i \Delta x),\ v_i(0) = 3 \end{cases}$$

$(i = 1, 2, ..., N = 500)$

Integration Interval : $[0, 10]$

| Brusselator | $ATOL = 10^{-90}, RTOL = 10^{-25}$ | | | |
|---|---|---|---|---|
| # stages($m$) | 20 | | | |
| Precisions | DD | MPFR(106) | QD | MPFR(212) |
| Comp.Time(s) | 7368.0 | 5794.7 | 55020.7 | 7974.1 |
| # steps | 1336 | 1354 | 1305 | 1272 |
| Average (s) | 5.5 | 4.3 | 42.2 | 6.3 |
| Max.Rel.Error | 1.1E-19 | 3.8E-20 | 8.4E-19 | 8.1E-19 |




- The discretization of $x$ direction is fixed for benchmarking.
- Double precision left preconditioned BiCGSTAB method in DP-MP mixed precision iterative refinement.

# C) 悪条件代数方程式の求解

# 実係数代数方程式の固有値解法

$$p_n(x) = 0 \quad \text{ここで } p_n(x) = \sum_{i=0}^{n} a_i x^i \ (a_i \in \mathbb{R}, a_n \neq 0)$$

$$q_n(x) = x^n + \sum_{i=0}^{n-1} c_i x^i \ (c_i = a_i / a_n) \Rightarrow C_n = \begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & 1 & 0 \\ 0 & \cdots & \cdots & 0 & 1 \\ -c_0 & -c_1 & \cdots & -c_{n-2} & -c_{n-1} \end{bmatrix}$$

xGEEVで固有値のみ求める

# 代数方程式向け同時反復法

- 実装と並列化が簡単。
- 解が近接するほど，計算精度を上げる必要がある。
- 固有値問題としては良条件でも，代数方程式としては悪条件になることが多い。

$$\mathbf{z}_k = [z_1^{(k)} \ z_2^{(k)} \ \dots \ z_n^{(k)}]^T \in \mathbb{C}^n$$

**2 次解法**

$$z_i^{(k+1)} := z_i^{(k)} - \frac{q_n(z_i^{(k)})}{\prod_{j=1, j\neq i}^{n}(z_i^{(k)} - z_j^{(k)})}$$

**3 次解法**

$$z_i^{(k+1)} := z_i^{(k)} - \frac{\frac{p_n(z_i^{(k)})}{p_n'(z_i^{(k)})}}{1 - \frac{p_n(z_i^{(k)})}{p_n'(z_i^{(k)})} \sum_{\substack{j=1 \\ j\neq i}}^{n} (z_i^{(k)} - z_j^{(k)})^{-1}}$$

# 混合精度計算による高速化

1. DD精度のxGEEV(Rgeev)でコンパニオン行列Cの固有値を求める
2. これを同時反復法の初期値とし，係数以上の精度で反復する

- Abirthの初期値を比較対象として使用。

$$z_i^{(0)} := -\frac{c_{n-1}}{n} + r\exp\left\{\left(\frac{2(i-1)\pi}{n} + \frac{3}{2n}\right)\mathrm{i}\right\}$$

$$r := \max_{0\le i\le(n-1)} |n_{\mathrm{nz}} c_i|^{1/(n-i)} \qquad \text{非零係数数 } n_{\mathrm{nz}} \le n$$

- 小澤の初期値も試みたが，高速化できるものとそうではないものがあり，DD精度固有値の初期値が最も良かった。
- 停止条件：

$$|z_i^{(k+1)} - z_i^{(k)}| \le \varepsilon_{\mathrm{rel}} |z_i^{(k)}| + \varepsilon_{\mathrm{abs}}$$

# 悪条件（実係数）代数方程式の例：全て重解なし

- Wilkinsonの例題：全て実数解

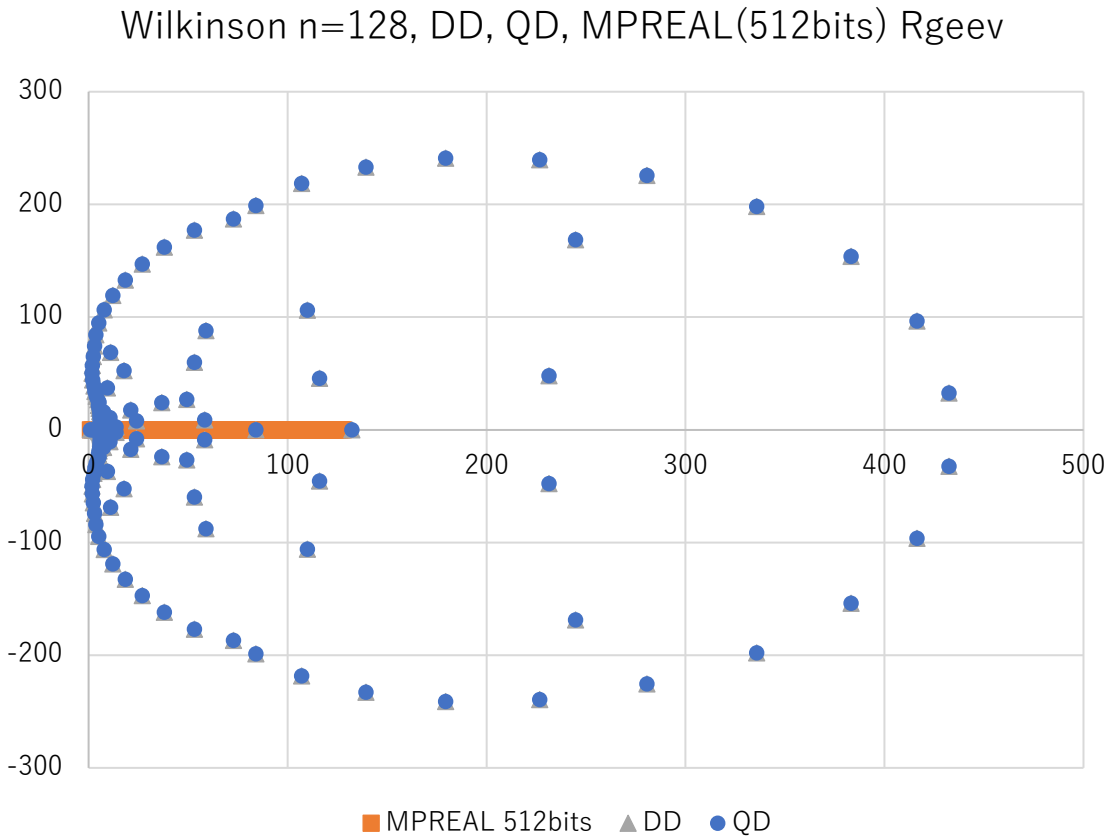
$\alpha_i = i$ となる $p_n(x) = \prod_{i=1}^{n}(x-i)$ を展開して与える

- Chebyshev積分公式の分点：ほぼ全て共役複素数解

$$a_n = 1 \Rightarrow \begin{cases} a_{n-(2k-1)} := 0 \\ a_{n-2k} := -\sum_{2j+1}^{k} a_{n-2(k-j)} \end{cases}$$

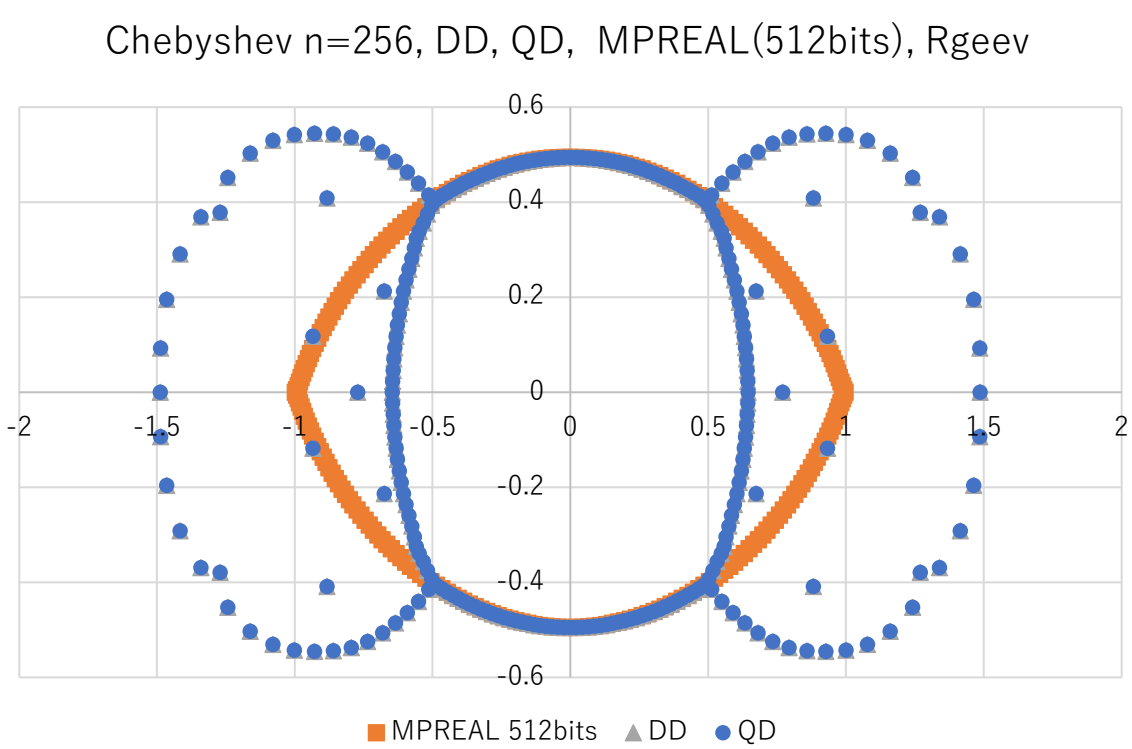
ここで，$k = 1, 2, ..., \lfloor n/2 \rfloor$ である．

# Wilkinsonの例題: $n = 128$




- $a_0 = 3.8\cdots \times 10^{215}$
- 係数は512bits計算
- コンパニオン行列をDD, QD, MPREAL 512bits精度で与えてRgeevを適用
- DD, QD精度は全く精度が出ていない→初期誤差の影響
- MPREAL 512bits計算でようやく実数解が現れる

# Chebyshev積分公式の分点計算: $n = 256$




- 係数は512bits計算
- コンパニオン行列をDD, QD, MPREAL 512bits精度で与えてRgeevを適用
- DD, QD精度は全く精度が出ていない→初期誤差の影響
- MPREAL 512bits計算でようやく精度の良い複素数解が現れる

# Wilkinsonの例題: n=128, 512bits計算

n=128 Wilkinson $\varepsilon_{\mathrm{rel}} := 7.5 \times 10^{-145}, \varepsilon_{\mathrm{abs}} := 1.0 \times 10^{-300}$

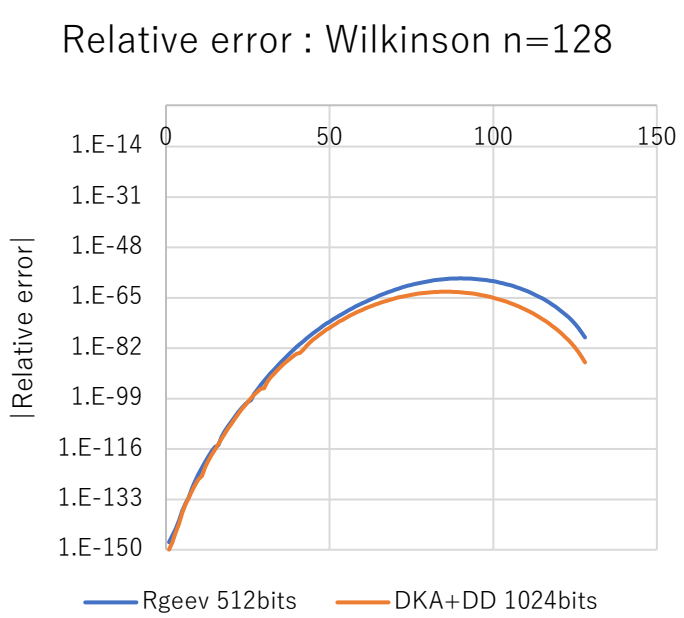
| EPYC | MPLAPACK Rgeev | | DKA2(1024bits) | | DKA3(1024bits) | | DK2(1024bits)+DD | | DK3(1024bits)+DD | |
|---|---|---|---|---|---|---|---|---|---|---|
| Unit: second | DD | MPFR 512bits | Second | #Iter. | Second | #Iter. | Second | #Iter. | Second | #Iter. |
| 1 Thread | 0.106 | 5.1 | 79.7 | 1374 | 71.1 | 691 | 30.8 | 538 | 51.5 | 512 |
| 2 Threads | | | 39.8 | 1374 | 36.6 | 691 | 15.4 | 535 | 26.1 | 509 |
| 4 Threads | | | 20 | 1374 | 18.7 | 691 | 8.14 | 562 | 14 | 512 |
| 8 Threads | | | 10.1 | 1374 | 8.96 | 691 | 1.43 | 195 | 1.42 | 99 |
| 16 Threads | | | 5.15 | 1374 | 4.55 | 691 | 2.07 | 557 | 3.32 | 511 |
| 24 Threads | | | 3.88 | 1374 | 3.74 | 691 | 1.52 | 541 | 2.69 | 511 |

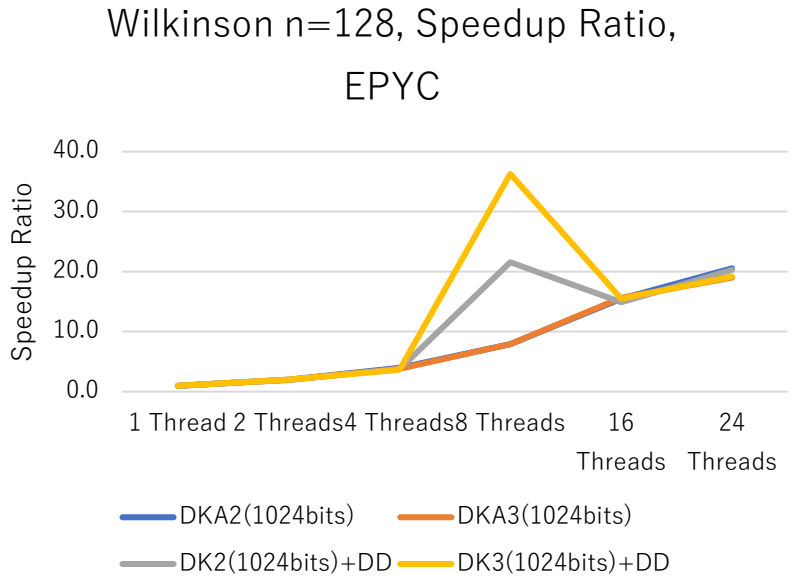
n=128 Wilkinson

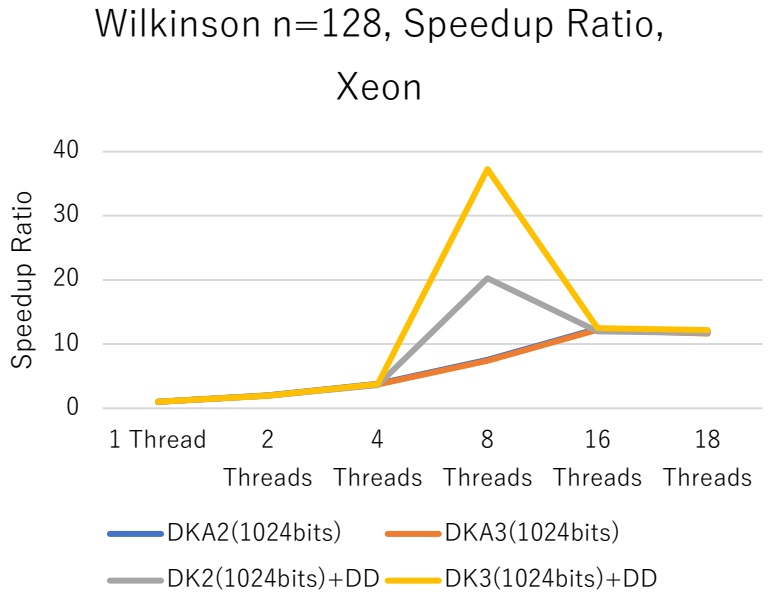
| Xeon | MPLAPACK Rgeev | | DKA2(1024bits) | | DKA3(1024bits) | | DK2(1024bits)+DD | | DK3(1024bits)+DD | |
|---|---|---|---|---|---|---|---|---|---|---|
| Unit: second | DD | MPFR 512bits | Second | #Iter. | Second | #Iter. | Second | #Iter. | Second | #Iter. |
| 1 Thread | 0.09 | 3.5 | 58.6 | 1374 | 50.8 | 691 | 22.3 | 538 | 37.1 | 512 |
| 2 Threads | | | 29.2 | 1374 | 26.2 | 691 | 11.3 | 535 | 18.8 | 509 |
| 4 Threads | | | 15.3 | 1374 | 13.7 | 691 | 6.18 | 562 | 9.93 | 512 |
| 8 Threads | | | 7.78 | 1374 | 6.86 | 691 | 1.1 | 195 | 0.996 | 99 |
| 16 Threads | | | 4.73 | 1374 | 4.16 | 691 | 1.86 | 557 | 2.97 | 511 |
| 18 Threads | | | 4.88 | 1374 | 4.37 | 691 | 1.9 | 557 | 3.05 | 511 |

赤字部分はMPFR Rgeev計算時間より高速であることを示す。

# Wilkinsonの例題: EPYC vs. Xeon




- MPFR Rgeevと同時反復法は同程度の相対誤差の近似値を得ている。
- 8 Threadsで効率が急上昇しているのは, DD精度の近似値を用いた場合の反復回数が激減したことによる。

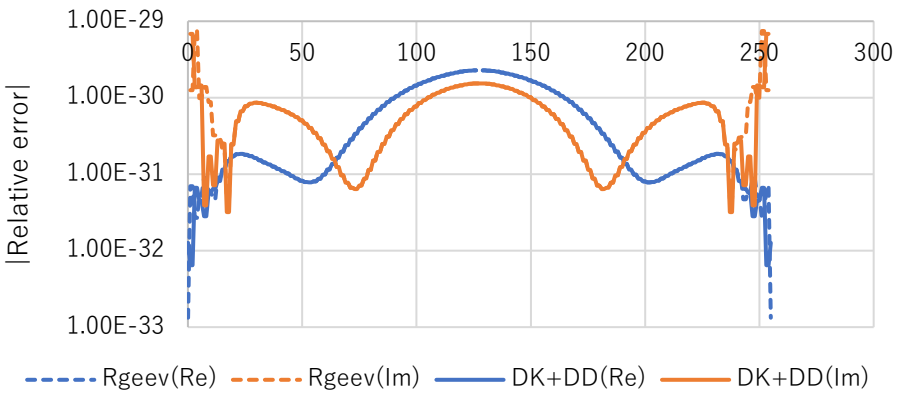
# Chebyshev積分公式の分点: n=256, 256 bits係数

n=256 Chebyshev

$\varepsilon_{\mathrm{rel}} := 8.6 \times 10^{-68},\ \varepsilon_{\mathrm{abs}} := 1.0 \times 10^{-300}$

| EPYC | MPLAPACK Rgeev | | DKA2(512bits) | | DKA3(512bits) | | DK2(512bits)+DD | | DK3(512bits)+DD | |
|---|---|---|---|---|---|---|---|---|---|---|
| Unit: second | DD | MPFR 256bits | Second | #Iter. | Second | #Iter. | Second | #Iter. | Second | #Iter. |
| 1 Thread | 1.29 | 57.6 | 203.0 | 1344 | 183 | 671 | 47 | 316 | 70.3 | 264 |
| 2 Threads | | | 100.0 | 1344 | 93.8 | 671 | 7.11 | 96 | 10 | 73 |
| 4 Threads | | | 51.1 | 1344 | 46.3 | 671 | 3.62 | 96 | 5.08 | 71 |
| 8 Threads | | | 25.3 | 1344 | 23.1 | 671 | 5.77 | 308 | 8.92 | 264 |
| 16 Threads | | | 13 | 1344 | 11.7 | 671 | 0.918 | 95 | 1.22 | 71 |
| 24 Threads | | | 8.96 | 1344 | 8.1 | 671 | 2.02 | 263 | 3.1 | 263 |

Chebyshev n=256, 256bits-Rgeev, 512bits-DK



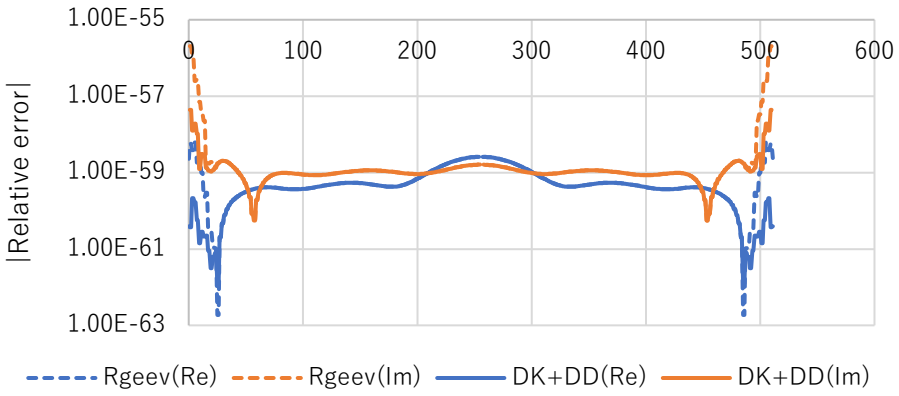

- Rgeevと同時反復法の近似解はほぼ同じ相対誤差の近似解を得ている。
- DK2-DD, DK3-DD(DD Rgeevを初期値とする)は反復回数の変化が大きい。
- 赤字部分はMPFR Rgeev計算時間より高速であることを示す。

# Chebyshev積分公式の分点: n=512, 512bits係数

n=512 Chebyshev $\varepsilon_{\mathrm{rel}} := 7.5 \times 10^{-145}$, $\varepsilon_{\mathrm{abs}} := 1.0 \times 10^{-300}$

| EPYC | MPLAPACK Rgeev | | DKA2(1024bits) | | DKA3(1024bits) | | DK2(1024bits)+DD | | DK3(1024bits)+DD | |
|---|---|---|---|---|---|---|---|---|---|---|
| Unit: second | DD | MPFR 512bits | Second | #Iter. | Second | #Iter. | Second | #Iter. | Second | #Iter. |
| 1 Thread | 9.84 | 636.0 | 2850.0 | 3032 | 2460 | 1510 | 500 | 552 | 577 | 357 |
| 2 Threads | | | 1370.0 | 3032 | 1210 | 1510 | 90.9 | 200 | 67.8 | 82 |
| 4 Threads | | | 689 | 3032 | 614 | 1510 | 46.1 | 202 | 34.5 | 83 |
| 8 Threads | | | 344 | 3032 | 307 | 1510 | 62.8 | 550 | 77.3 | 363 |
| 16 Threads | | | 173 | 3032 | 174 | 1510 | 11.8 | 205 | 8.59 | 84 |
| 24 Threads | | | 120 | 3032 | 106 | 1510 | 8.47 | 213 | 6.57 | 83 |

Chebyshev n=512, 512bits-Rgeev, 1024bits-DK




- Rgeevと同時反復法の近似解はほぼ同じ相対誤差の近似解を得ている。
- DK2-DD, DK3-DD(DD Rgeevを初期値とする)は反復回数の変化が大きい。
- 赤字部分はMPFR Rgeev計算時間より高速であることを示す。

# Chebyshev積分公式の分点：Xeon

n=256

| Xeon | MPLAPACK Rgeev | | DKA2(512bits) | | DKA3(512bits) | | DK2(512bits)+DD | | DK3(512bits)+DD | |
|---|---|---|---|---|---|---|---|---|---|---|
| Unit: second | DD | MPFR 256bits | Second | #Iter. | Second | #Iter. | Second | #Iter. | Second | #Iter. |
| 1 Thread | 1.12 | 38.4 | 146.0 | 1344 | 127 | 671 | 32.1 | 316 | 49.2 | 264 |
| 2 Threads | | | 69.0 | 1344 | 64.8 | 671 | 4.95 | 96 | 6.91 | 73 |
| 4 Threads | | | 36.3 | 1344 | 34.2 | 671 | 2.61 | 96 | 3.59 | 71 |
| 8 Threads | | | 18.6 | 1344 | 17.7 | 671 | 4.2 | 308 | 6.63 | 264 |
| 16 Threads | | | 10.9 | 1344 | 10 | 671 | 0.779 | 95 | 1.07 | 71 |
| 18 Threads | | | 10.8 | 1344 | 9.79 | 671 | 2.32 | 293 | 3.62 | 250 |

n=512

| Xeon | MPLAPACK Rgeev | | DKA2(1024bits) | | DKA3(1024bits) | | DK2(1024bits)+DD | | DK3(1024bits)+DD | |
|---|---|---|---|---|---|---|---|---|---|---|
| Unit: second | DD | MPFR 512bits | Second | #Iter. | Second | #Iter. | Second | #Iter. | Second | #Iter. |
| 1 Thread | 9.25 | 441.0 | 2030.0 | 3032 | 1740 | 1510 | 362 | 552 | 407 | 357 |
| 2 Threads | | | 1020.0 | 3032 | 937 | 1510 | 67.3 | 200 | 48.2 | 82 |
| 4 Threads | | | 528 | 3032 | 471 | 1510 | 35.3 | 202 | 25.5 | 83 |
| 8 Threads | | | 267 | 3032 | 238 | 1510 | 48.5 | 550 | 56.5 | 363 |
| 16 Threads | | | 159 | 3032 | 142 | 1510 | 10.8 | 205 | 7.85 | 84 |
| 18 Threads | | | 151 | 3032 | 132 | 1510 | 26.9 | 539 | 31.7 | 365 |

# IV. 今後の課題

# やり残しを片付ける＆ライブラリ整備

- やり残しを片付ける
  - 疎行列（SpMV）実装→Lanczos法の高速化
  - BNCmatmulベースの陰的Runge-Kutta法の再実装
  - 混合精度代数方程式の求解→行列の固有値問題の高速化
- ライブラリ整備
  - WASMを用いたWeb上の多倍長精度計算の高速化
  - Pythonモジュールの整備
  - ドキュメント整理