数値解析 2
第2回

静岡理工科大学
情報学部　コンピュータシステム学科
幸谷智紀
https://na-inet.jp/mpna/

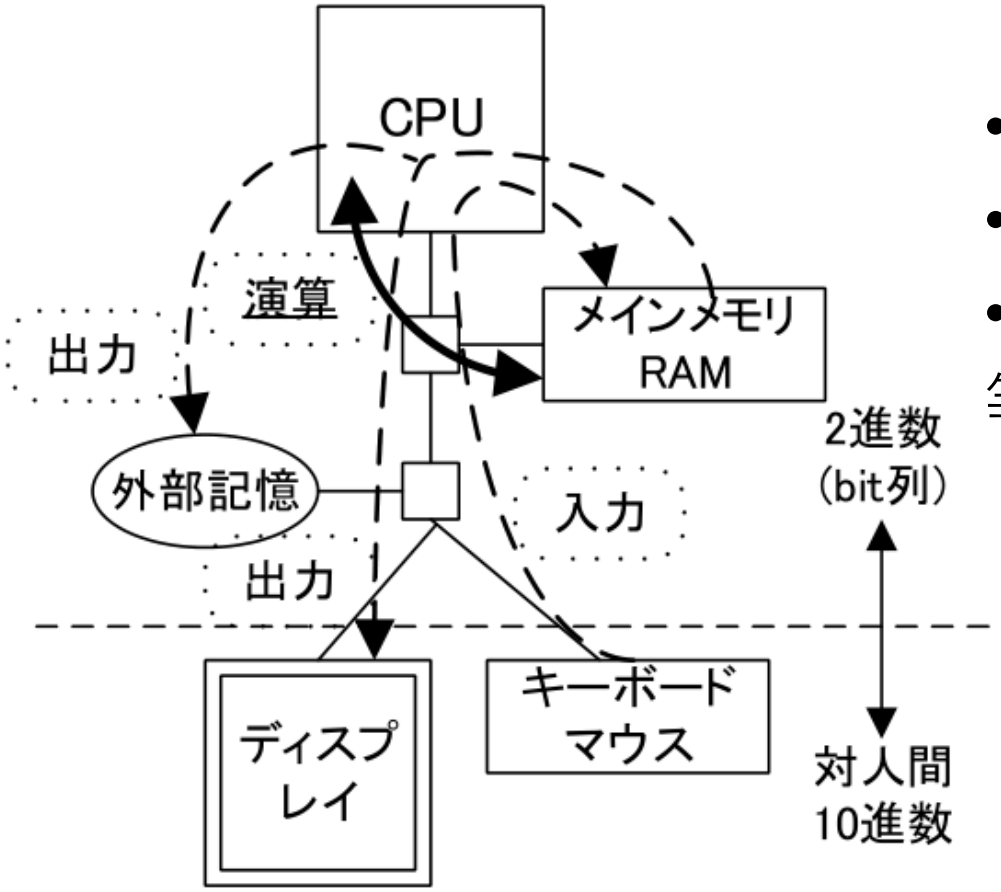
シラバス

1. 多倍長数値計算とは？
2. コンピュータの内部と演算の高速化手法
3. 多倍長自然数演算と多倍長整数演算
4. 多倍長整数演算とその応用
5. 多倍長有理数演算とその応用
6. GMPを用いたRSA暗号の実装

→中間レポート

7. GMPのMPF型とMPFR
8. マルチコンポーネント型多倍長演算の実装
9. QDライブラリの利用法
10. 基本線型計算
11. 連立一次方程式の求解(1/2)
12. 連立一次方程式の求解(2/2)
13. べき乗法と逆べき乗法の実装(1/2)
14. べき乗法と逆べき乗法の実装(2/2)
15. 講義のまとめと最終試験

コンピュータの内部と演算の高速化手法



- **高速な（新しい）ハードウェアを使用する**
- **並列化する(SIMD, OpenMP, MPI)**
- **演算量を減らす**
- **キャッシュメモリの有効利用**

等々

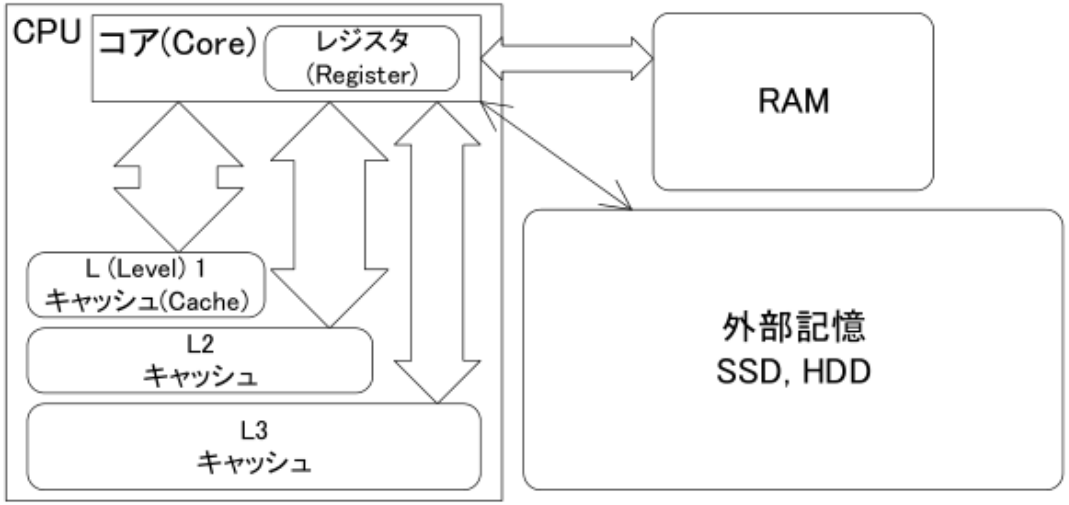


図 2.1　PC の HW アーキテクチャ (左) とメモリ階層 (右)

行列乗算の高速化

行列の乗算 (matrix multiplication) はハードウェアの演算性能を確認するためによく使用される計算事例である。理由としては

- 線形代数の基礎知識があれば誰でも理解できる程度に計算が簡単
- 行列サイズを大きく取ることでメインメモリメモリを目いっぱい使用することができる
- 並列化が容易であるため，マルチコア構成の CPU や，GPU の性能を最大限発揮できる
- キャッシュメモリのヒット率を最大限上げることで，RAM から CPU へのアクセスを減らし，演算性能の向上が期待できる

といったことが挙げられる。本章では n 次の実正方行列 $\mathbb{R}^{n\times n}$ の乗算を例に，高速化の事例を紹介する。

以降扱う正方行列 $A, B \in \mathbb{R}^{n\times n}$ を次のように指定する。各行列の要素を a_{ij}, b_{ij} と書き，

$$a_{ij} = \sqrt{5}\,(i+j-1),\ b_{ij} = \sqrt{3}\,(n-(i+j-2))$$

とする。これを用いて行列 A, B の積 $C := AB$ を求める。この時，$C \in \mathbb{R}^{n\times n}$ は

$$c_{ij} := \sum_{k=1}^{n} a_{ik} b_{kj} \tag{3.1}$$

となる。

matmul_simple.cpp(1/5)

- 正方行列の大きさ：n = dim
- れぞれの行列要素を 1 次元配列 mat_a, mat_b に行優先 (行方向に要素を順に入れていく方式) 代入
- 定義式 (3.1) 通りに計算し，積を ret に代入して返す関数 matmul_simple(14 ～ 30 行目) を含む，行列乗算ベンチマークテストプログラム

- コンパイル方法
  1. getsecv.h, getsec.cpp, matmul_block.h, matmul_simple.cppをダウンロードして同じフォルダ(ディレクトリ)に格納
  2. $ g++ matmul_simple.cpp –lm –o matmul_simple
  3. $ ./matmul_simple

matmul_simple.cpp(2/5)

```cpp
#include <iostream>
#include <iomanip>
#include <cmath>

// matmul_gflops関数, byte_double_sqmt関数, normf_dmatrix_array関数
#include "matmul_block.h"

// 時間計測用: get_secv関数, get_real_secv関数
#include "get_secv.h"

using namespace std;

// 正方行列×正方行列
// 行優先方式で値を格納
void matmul_simple(double ret[], double mat_a[], double mat_b[], int dim)
{
  int i, j, k, ij_index;

  for(i = 0; i < dim; i++)
  {
    for(j = 0; j < dim; j++)
    {
      ij_index = i * dim + j; // 行優先方式
      ret[ij_index] = 0.0;
      for(k = 0; k < dim; k++)
        ret[ij_index] += mat_a[i * dim + k] * mat_b[k * dim + j];
    }
  }
}
```

- matmul_block.hに必要な行列乗算，GFLOPs計算，行列のバイト数，フロベニウスノルム等を定義。

$$\|C\|_F = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{n}|c_{ij}|^2}$$

- get_secv.hに時間計測用関数を定義
- ここでは単純行列乗算アルゴリズムを使用してmatmul_simple関数を実装。

matmul_simple.cpp(3/5)

```
31  // メイン関数
32  int main(int argc, char *argv[])
33  {
34    int i, j, min_dim, max_dim, dim, iter, max_iter = 10, num_threads;
35    double *mat_a, *mat_b, *mat_c;
36    double stime, etime;
37
38    if(argc < 3)
39    {
40      cout << "Usage:␣" << argv[0] << "␣[min.␣dimension]␣␣[max.dimension]"<< endl;
41      return 1;
42    }
43
44    min_dim = atoi(argv[1]);
45    max_dim = atoi(argv[2]);
46
47    if(min_dim <= 0)
48    {
49      cout << "Illegal␣dimension!␣(min_dim␣=␣" << min_dim << ")" << endl;
50      return 1;
51    }
```

matmul_simple.cpp(4/5)

```
53    // メインループ
54    cout << setw(5) << "␣␣dim␣:␣␣␣␣␣SECONDS␣GFLOPS␣Mat.KB␣||C||_F" << endl;
55    for(dim = min_dim; dim <= max_dim; dim += 16)
56    {
57
58      // 変数初期化
59      mat_a = (double *)calloc(dim * dim, sizeof(double));
60      mat_b = (double *)calloc(dim * dim, sizeof(double));
61      mat_c = (double *)calloc(dim * dim, sizeof(double));
62
63      // mat_aとmat_bに値入力
64      for(i = 0; i < dim; i++)
65      {
66        for(j = 0; j < dim; j++)
67        {
68          mat_a[i * dim + j] = sqrt(5.0) * (double)(i + j + 1);
69          mat_b[i * dim + j] = sqrt(3.0) * (double)(dim - (i + j));
70        }
71      }
```

【重要】「行列乗算の時間計測部分

```
92      // 変数消去
93      free(mat_a);
94      free(mat_b);
95      free(mat_c);
96
97    } // メインループ終了
98
99    return 0;
100 }
```

matmul_simple.cpp(5/5)

```
73      max_iter = 3; // 行列乗算を最低3回実行
74
75      do
76      {
77        stime = get_real_secv();
78        for(iter = 0; iter < max_iter; iter++)
79          matmul_simple(mat_c, mat_a, mat_b, dim);
80        etime = get_real_secv(); etime -= stime;
81
82        if(etime >= 1.0) break; // 1秒以上になるまで繰り返し
83
84        max_iter *= 2;
85      } while(0);
86
87      etime /= (double)max_iter; // 平均計算時間を導出
88
89      // 出力
90      cout << setw(5) << dim << "␣:␣" << setw(10) << setprecision(5) << etime << "␣" <<
            matmul_gflops(etime, dim) << "␣" << byte_double_sqmat(dim) / 1024 << "␣" <<
            normf_dmatrix_array(mat_c, dim, dim) << endl;
```

- 行列乗算(matmul_simple関数)の平均実行時間を計測し，表示する。
- 最低でも3階以上は必ず実行して平均値を取る。
- 可能ならば，標準偏差もあると良い。
- 行列乗算結果はフロベニウスノルムを使用

OpenMPによるメモリ共有の並列化(1/3)

1. OpenMP 有効時 (OPENMP 定義がある場合は有効) のみ， OpenMP 関数を使うためのヘッダファイル omp.h をインクルードする。

```
#ifdef _OPENMP // OpenMP使用時のみ有効
  #include <omp.h> // (1) OpenMP関数
#endif // _OPENMP
```

2. #pragma omp ディレクティブを matmul simple 関数内に挿入し，複数スレッドで for ループ内の処理を分担させる。ここでは ij index と k は各スレッド内のプライベート変数，即ち，スレッドごとに独立した変数として指定し，互いのスレッド内処理が干渉しないようにしている。

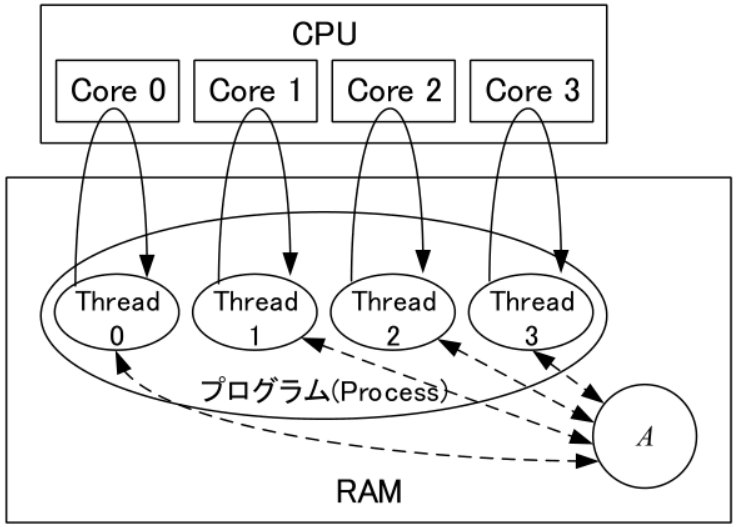


図 3.3　マルチコア CPU

キャッシュメモリの有効活用

正方行列 A, B を同じサイズの小行列に分割し，小行列単位の計算に分割して行列乗算を行う計算法をブロック化 (blocking) アルゴリズム，またはタイリング (tiling) と呼ぶ。これはキャッシュヒット率を上げるための工夫である。行列乗算の場合，

$$A = [A_{ik}], B = [B_{kj}] \ (1 \leq i \leq M, 1 \leq k \leq L, 1 \leq j \leq N)$$

のように A, B をブロック行列化して，行列 $C = [C_{ij}]$ を次のように計算する。

$$C_{ij} := \sum_{k=1}^{L} A_{ik} B_{kj}$$

例えば，$M = L = N = 4$ と分割すると図 3.2 のようになる。このとき分割された小行列の大きさをブロックサイズと呼び，この場合は $n_{\min} \times n_{\min}$ である。

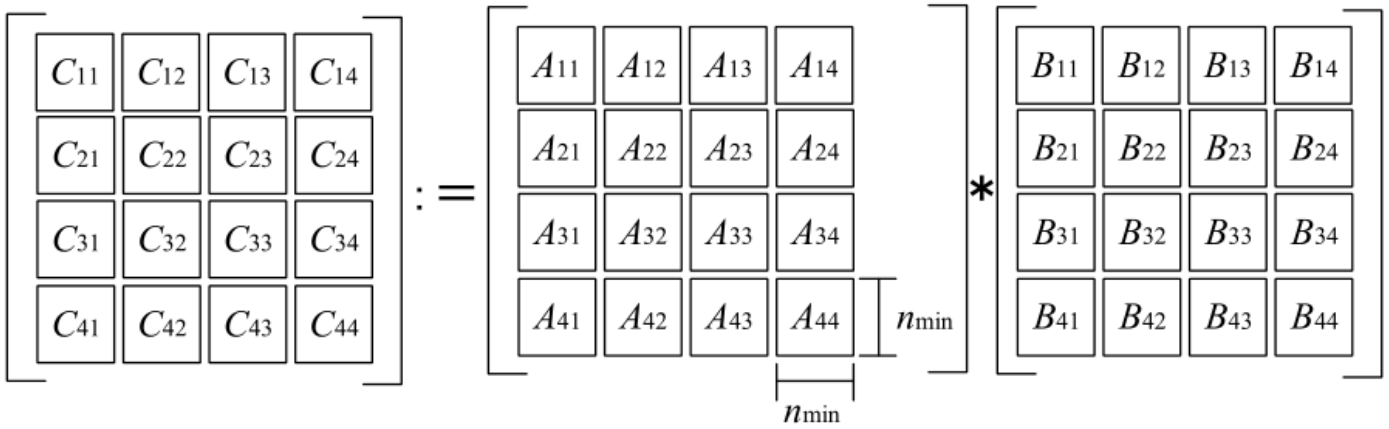


図 3.2　ブロック化アルゴリズム

演算量を減らす：Strassenのアルゴリズム

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix},\ B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \qquad C := \begin{bmatrix} P_1 + P_4 - P_5 + P_7 & P_3 + P_5 \\ P_2 + P_4 & P_1 + P_3 - P_2 + P_6 \end{bmatrix}$$

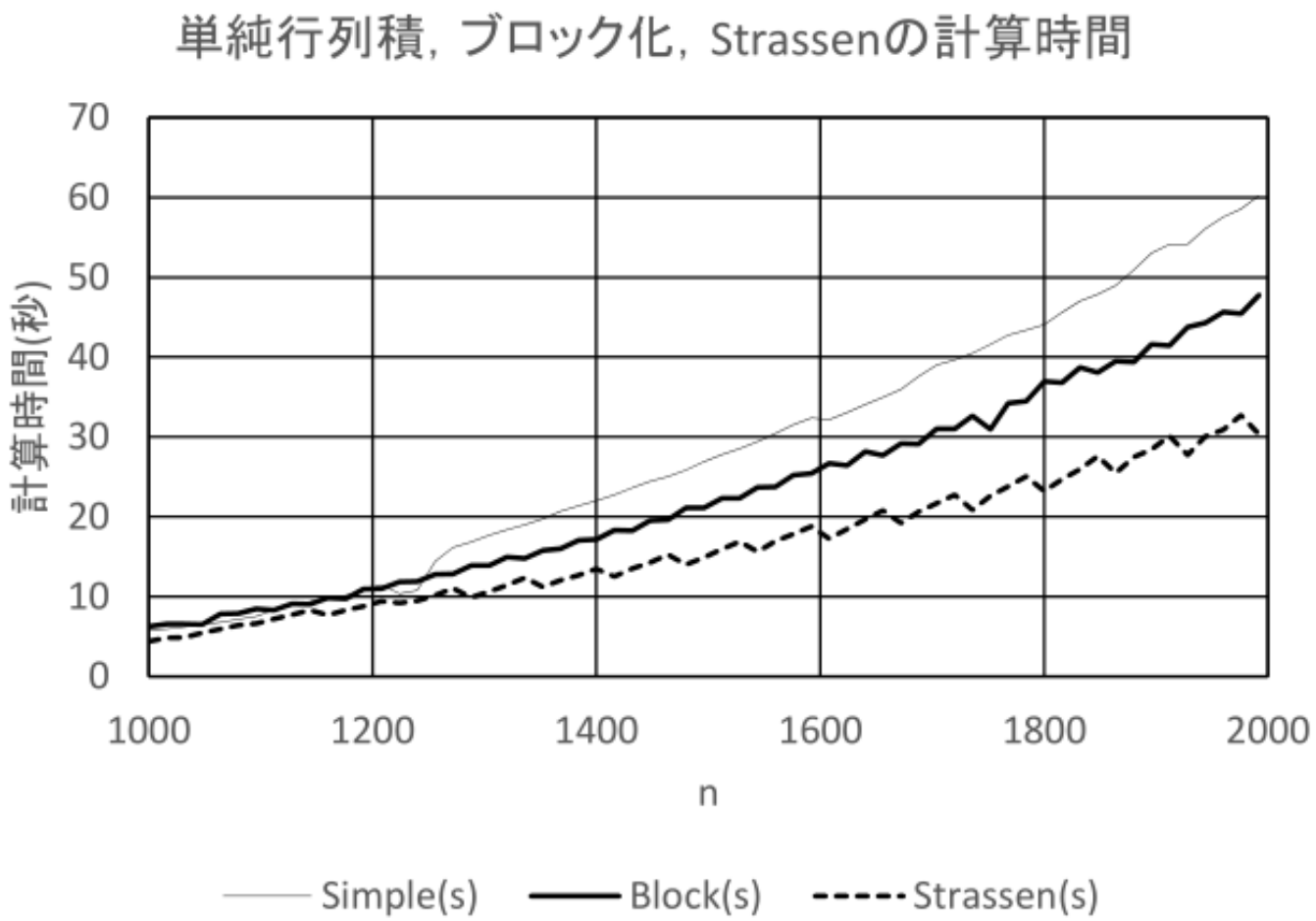
$$\begin{aligned} P_1 &:= (A_{11} + A_{22})(B_{11} + B_{22}) \\ P_2 &:= (A_{21} + A_{22})B_{11} \\ P_3 &:= A_{11}(B_{12} - B_{22}) \\ P_4 &:= A_{22}(B_{21} - B_{11}) \\ P_5 &:= (A_{11} + A_{12})B_{22} \\ P_6 &:= (A_{21} - A_{11})(B_{11} + B_{12}) \\ P_7 &:= (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned}$$

Strassen のアルゴリズムを適用することにより，n 次正方行列の行列積に必要となる乗算量 $M(n)$ と加減算 $A(n)$ とすると，

$$M(n) = 7M(n/2),\ A(n) = 7A(n/2) + 18(n/2)^2 \tag{3.6}$$

となる。これを再帰的に適用していくことにより，結果として演算量が通常の行列乗算アルゴリズムより減らすことができる。究極的にはトータルの演算量が $n^{\log_2 7}$ に比例するものになる。単純行列乗算はブロック化アルゴリズムの演算量は n^3 に比例するのに対して，大行列になればなるほど効果が高い。

Strassen > ブロック化 > 単純行列乗算



OpenMPによるメモリ共有の並列化(2/3)

```
// 正方行列×正方行列
// 行優先方式で値を格納
void matmul_simple(double ret[], double mat_a[], double mat_b[], int dim)
{
. . .
  for(i = 0; i < dim; i++)
  {
    #pragma omp parallel for private(ij_index, k) // (2) OpenMPディレクティブ
    for(j = 0; j < dim; j++)
    {
      ij_index = i * dim + j;
      ret[ij_index] = 0.0;
      for(k = 0; k < dim; k++)
        ret[ij_index] += mat_a[i * dim + k] * mat_b[k * dim + j];
    }
. . .
}
```

OpenMPによるメモリ共有の並列化(3/3)

```
int main(int argc, char *argv[])
{

  . . .

  if(min_dim <= 0)
  {
    cout << "Illegal␣dimension!␣(min_dim␣=␣" << min_dim << ")" << endl;
    return EXIT_FAILURE;
  }

#ifdef _OPENMP
  int num_threads;
  cout << "num_threads:␣";
  cin >> num_threads;

  omp_set_num_threads(num_threads);
#endif // _OPENMP

  // メインループ
  cout << setw(5) << "␣␣dim␣:␣␣␣␣␣SECONDS␣GFLOPS␣Mat.KB␣||C||_F" << endl;

  . . .

}
```

3. メイン関数で，起動するスレッド数を設定する。コア数以上のスレッドを起動しても並列処理は行われないので，必ずコア数以下のスレッド数を指定する必要がある。

本日の課題

1. 手近なマシン上で，単純行列乗算ベンチマークを行い，計算時間の比較を行え。また，GFLOPS 値での比較も行ってみよ。
2. OpenMPで並列化を行い，最低でも2スレッドで動作させ，高速化の効果が出るかどうかを確認せよ。

（注意）Excelで計算時間とGFLOPsのグラフを描画すること。

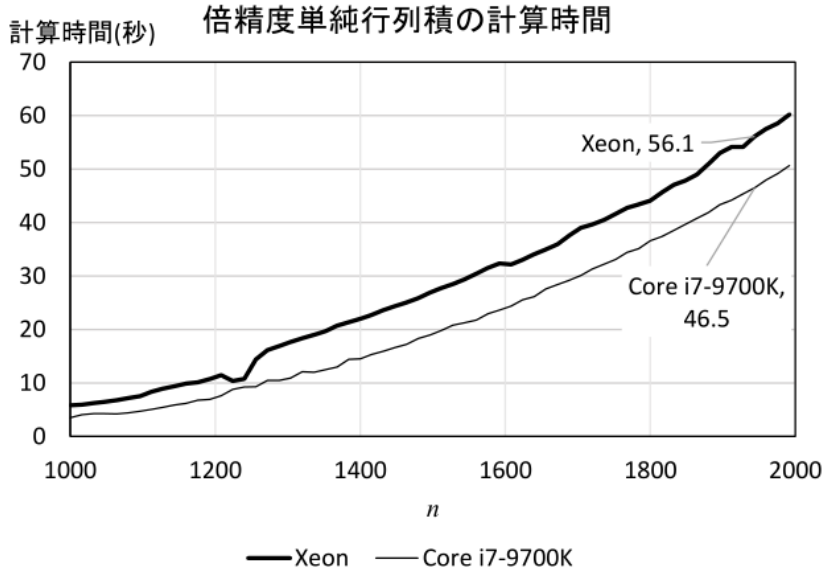


図 3.1　単純行列乗算の速度比較: Core i7-9700K vs. Xeon E5-2620

GFLOPsのグラフ
横軸：行列の大きさ(n)
縦軸：GFLOPs