Vol.2021-HPC-180 No.26
2021/7/21

LiNGAMを用いた大量変数の因果探索処理に向けた計算カーネルの高速化の検討

栗原 康志[1,a)]　川上 健太郎[1]　山崎 雅文[1]　松田 一仁[1]　山田 芙夕楓[1]　田原 司睦[1]　横山 乾[1]

概要：多次元ベクトルとして測定されたビッグデータから、ベクトルデータを構成する各変数間の因果 (主従) 関係を推定する統計的因果探索と呼ばれる処理がある。ベクトルデータの内、独立した変数成分が非ガウス分布であることを仮定できる場合、変数間の因果関係は LiNGAM(Linear Non-Gaussian Acyclic Model) アルゴリズムで解けることが知られている。Python を用いた LiNGAM の実装が OSS として公開されているが、現状は 1 台の PC で実行することを前提とされており、100 変数からなるデータの因果探索に約 200 秒、1000 変数で約 60 時間かかっている。我々は LiNGAM を適用可能とする領域を拡大するため、1 万変数以上を対象とする因果探索を目標にしている。LiNGAM の計算時間は変数の数の約 3 乗に比例するため、1 万変数以上を対象とする因果探索を行う場合，現状では数年〜数十年かかることが予想される。本稿では、1 万変数以上の因果探索を実用的な計算時間で行うために、計算カーネルの高速化の方法を検討したので、これを報告する。

A Study on speeding up Causal Search Process for Large Number of Variables Using LiNGAM

Abstract: From big data measured as multidimensional vectors, there is a process called statistical causal search that estimates the causal (principal-agent) relationship between each variable that constitutes the vector data. It is known that the causal relationship between variables can be solved by the Linear Non-Gaussian Acyclic Model (LiNGAM) algorithm if the independent variable components in the vector data can be assumed to be non-Gaussian. A Python implementation of LiNGAM has been released as OSS, but it is currently assumed to be run on a single PC. It takes about 200 seconds to perform causal search for data consisting of 100 variables, and about 60 hours for data consisting of 1000 variables. In order to expand the area of applicability of LiNGAM, we are aiming for a causal search targeting more than 10,000 variables. However, since the computation time of LiNGAM is proportional to the number of variables to the third power, it is currently expected to take several years to several 10 years. In this paper, in order to achieve causal inference for more than 10,000 variables in a practical computation time, we have investigated a method to speed up the computation kernel, and will report on this method in this paper.

1. はじめに

近年，医療現場における患者一人ひとりの発がんに影響を与える特徴的な遺伝子の特定や，マーケティングの現場における顧客一人ひとりの購入に繋がる特徴的な要因の特定といったことが求められている．この場合，属性 A と B の間に関連があるという相関関係だけでなく，A だから B であるという原因と結果を示した因果 (主従) 関係に着目する必要がある．因果関係を把握したい場合，観測データから全体の因果構造の推定を行う統計的因果探索が有効であり，ゲノミクス，臨床医学，疫学など幅広い分野に応用されている．

統計的因果探索は，多次元ベクトルとして測定されたデータから，ベクトルデータを構成する各変数間の因果関係を推定する．ベクトルデータの内，独立した変数成分が非ガウス分布であることを仮定できる場合, 変数間の因果関係は LiNGAM(Linear Non-Gaussian Acyclic Model)[1][4] アルゴリズムで解けることが知られている．LiNGAM は，Python を用いた実装が OSS[5] としてすでに公開されてお

[1] 富士通株式会社
4-1-1 Kamikodanaka, Nakahara-ku, Kawasaki, Kanagawa 211–8588, Japan
[a)] kouji3211@fujitsu.com

り，誰でも利用可能である．

現在，我々は LiNGAM を適用可能とする領域を拡大するため、1 万変数以上の大量変数を対象とする因果探索を行うことを目標にしている．しかし，OSS で公開されている LiNGAM の現状の実装は，1 台の PC で実行することを前提とされており，1 万変数以上の大量変数を対象にした因果探索を行うことは難しい状況にある．

そこで，我々はスーパーコンピュータ富岳 [6] の大規模実行環境を利用して，LiNGAM による 1 万変数以上の因果探索を実用的な計算時間で行う方法を検討している．

本稿では、そのうちの一つである LiNGAM の計算カーネルの高速化について検討結果と評価結果を述べる．

2. LiNGAM

2.1 LiNGAM モデル

LiNGAM モデルは，観測データが非ガウス分布に従うことを仮定した上で，構造式モデルに線形非巡回性を持たせたモデルである．p 個の観測変数 x_1，x_2，....，x_p が LiNGAM モデルに従って生成される場合，x_i は次のように書ける．

$$x_i = \sum_{j \neq i} b_i * x_j + e_i (i = 1, ..., p) \tag{1}$$

それぞれの観測変数 x_i は，その変数以外の観測変数 $x_j (j = 1, ..., p; j \neq i)$ とその誤差変数 e_i の線形和である．x_i の生成過程は有向非巡回グラフ (Directed Acyclic Graph, DAG) で表現することができ，このとき，b_{ij} は有向非巡回グラフの隣接行列にあたり，b_{ij} は変数 x_j から x_i への影響の強さを示し，係数 b_{ij} が 0 である場合，x_j から x_i への直接的な因果効果がないことを示す $(j = 1, ..., p; j \neq i)$．また，誤差変数 $e_i (i = 1, ..., p)$ は独立で，非ガウス連続分布に従う．この独立性は未観測共通原因がないことを意味する．式 (1) は，行列を使用すると次のように書ける．

$$\mathbf{x} = \mathbf{B}\mathbf{x} + \mathbf{e} \tag{2}$$

観測ベクトル $\mathbf{x}$ と誤差変数ベクトル $\mathbf{e}$ はそれぞれ，観測変数 x_i と誤差変数 $e_i (i = 1, ..., p)$ をまとめて表現している．また，正方行列 $\mathbf{B}$ は，係数 $b_{ij} (i, j = 1, ..., p)$ をまとめて表現しており，$\mathbf{B}$ を係数行列と呼ぶ．LiNGAM モデルは識別可能であるため，係数行列 $\mathbf{B}$ は観測変数の分布に基づいて，一意に推定可能である．

2.2 LiNGAM モデルの推定

LiNGAM モデルの係数行列 $\mathbf{B}$ を推定することにより，観測変数間の因果関係を明らかにすることができる．LiNGAM モデルの係数行列 $\mathbf{B}$ を推定するアプローチは大きく分けて二つ存在する．一つは独立成分分析の手法を用いる推定アプローチ (ICA-LiNGAM[1]) で，もう一つは，回帰分析と独立性評価を繰り返すアプローチ (Direct-LiNGAM[2][3]) である．どちらのアプローチも Python を用いた実装が OSS として公開されている．まずは，現在広く利用されている Direct-LiNGAM を用いて大規模変数の因果探索を行う場合について高速化の検討を行った．

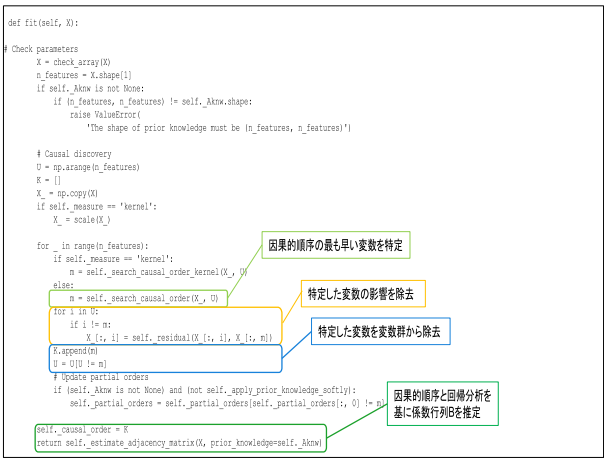

図 1 Direct-LiNGAM の fit 関数

Direct-LiNGAM では，以下の手順により係数行列 $\mathbf{B}$ を推定している．

(1) 観測変数群から 2 変数を取り出し，因果的順序の最も早い観測変数を特定
(2) 特定した観測変数の影響を取り除いて残りの変数の中でまた因果的順序の最も早い観測変数を特定
(3) 手順 (1),(2) を繰り返し実施して得られた因果的順序に従って回帰分析を行い，係数行列 $\mathbf{B}$ を推定

観測変数の因果的順序とは，その順序に従って変数を並び変えると，順序の後ろの変数が順序の前の変数の原因になることがない順序である．つまり，因果的順序が最も早い観測変数が因果の始まりの変数となる．

OSS として公開されている Direct-LiNGAM において，因果探索の処理は fit 関数として実装されている．fit 関数のソースコードを図 1 に示す．ソース中の_search_causal_order 関数で因果的順序の最も早い観測変数を特定 (手順 1) し，続く for ループにおいて特定した観測変数の影響を取り除く．次のイタレーションでは，特定した観測変数を取り除いた残りの観測変数群の中で再び因果的順序の最も早い観測変数を特定する (手順 2)．最後に_estimate_adjacency_matrix 関数を呼び出し，得られた因果的順序に従って回帰分析を行い，係数行列 $\mathbf{B}$ を推定する (手順 3)．

3. Direct-LiNGAM の現状課題と高速化に向けた検討

OSS として公開されている Direct-LiNGAM の現状の性能を評価した．評価には，Xeon Gold 6148 を CPU として搭載している Intel サーバーを利用した．Xeon Gold 6148

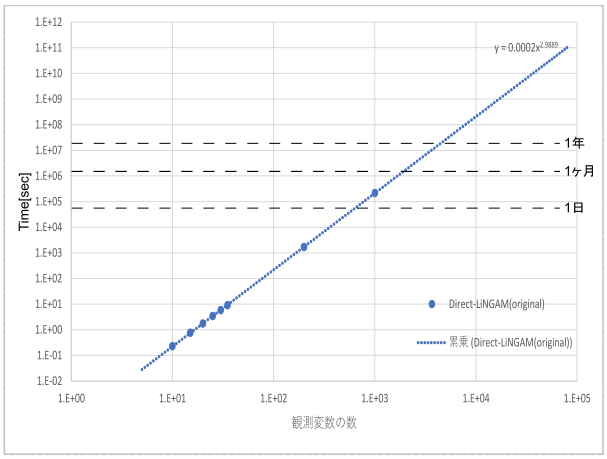

図 2 Direct-LiNGAM の現状性能

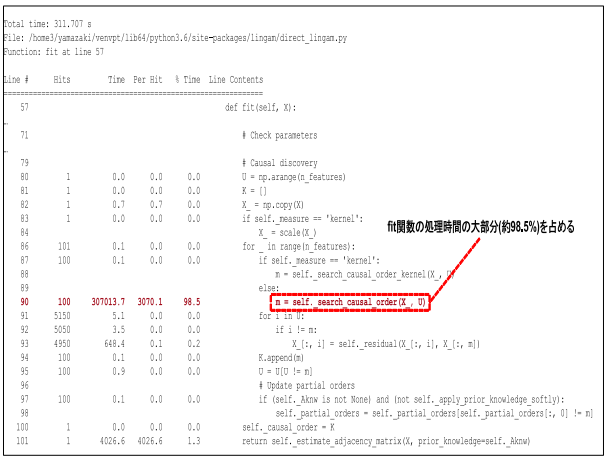

図 3 Line profile の取得結果 (fit 関数)

の動作周波数は 2.4GHz であるが，測定時にはターボブーストが ON の状態であったため，実際には 3.6GHz で動作している．また，Xeon Gold 6148 には複数の CPU コアが搭載されているが，現状の Direct-LiNGAM はシングルスレッドで動作するため，1 コアのみの動作である．測定結果を図 2 に示す．図 2 の横軸は観測変数の数を示し，縦軸は Direct-LiNGAM の実行時間 [sec] を示している．

100 変数からなる観測データの因果探索を行った場合，約 200 秒を要し，1000 変数からなる観測データの因果探索を行った場合，約 60 時間を要していた．性能測定結果から実行時間の近似直線を求めると，$O(x^3)$ に近い近似直線が得られた．1 万変数以上の観測データについて因果探索を行った場合の実行時間を近似曲線から求めると，数年〜数十年を要することになる．従って，1 万変数以上の観測データの因果探索に現状の Direct-LiNGAM を利用することは現実的に困難であり，100,000〜1,000,000 倍の性能改善が必要であると考えられる．

まず，現状の Direct-LiNGAM の性能改善に向けて，fit 関数の Line profile を取得し、fit 関数内の処理時間の内訳について調査した．結果を図 3 に示す．fit 関数の処理時間の大部分は，因果順序を探索する_search_causal_order 関数が占めていることが分かった．次に，_search_causal_order 関数内の処理時間の内訳を図 4 に示す．_search_causal_order 関数の中は，2 重ループの処理になっている．このループの処理が_search_causal_order 関数の処理時間の大部分を占めており，高速化が必要な個所であることが分かった．

そこで，_search_causal_order 関数に関して 3 つの高速化施策を検討している (図 5)．一つ目はシングルスレッドあたりの処理の高速化である．図 4 の結果を見ると，_diff_mutual_info 関数の処理時間も大きいことが分かる．そこで，内部で行われている複数の算術関数の処理 (計算カーネル) についてフュージョンした関数を JIT 技術により生成することで高速化を行う (図 5 中の (1))．二つ目と三つ目はループの並列処理化である．最外の Uc のループ

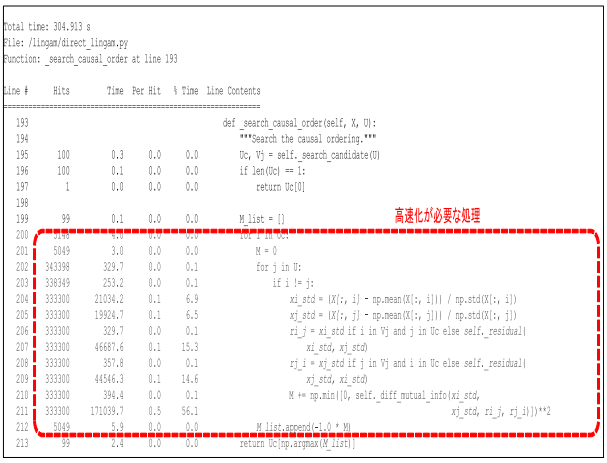

図 4 Line profile の取得結果 (_search_causal_order 関数)

において各イタレーション間は独立した演算である．そのため，並列に処理しても結果が変わらない．従って，各イタレーションの処理を富岳の複数の計算ノードで同時に実行することにより高速化を図ることが可能である (図 5 中の (2))．また，富岳の計算ノードは複数の CPU コアを搭載しているため，計算ノード内でも並列処理を行うことによりさらに高速化可能であると考えている (図 5 中の (3))．

本稿では，一つ目のシングルスレッドあたりの処理の高速化に当たる計算カーネルの高速化について検討した結果を以降の章で報告する．

4. 計算カーネルの高速化

4.1 高速化対象の計算カーネル

_diff_mutual_info 関数の処理において，多くの時間を要している処理は def_entropy 関数である．def_entropy 関数のコードを図 6 に示す．def_entropy 関数は，配列の要素に対して算術演算を繰り返すような処理を持つ関数である．例えば，$np.log(np.cosh(u))$ の項では，配列 u の各要素に対して cosh を計算し，その各結果に対して log を計算する．$u * np.exp((-u**2)/2)$ の項では，配列 u の各要素に

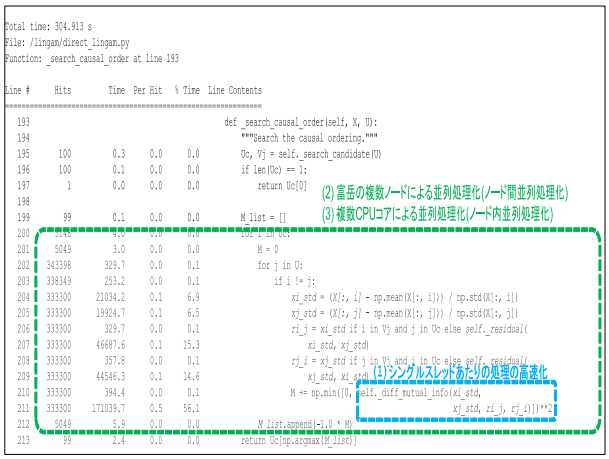

図 5 高速化施策の検討

```
def _entropy(self,u)
# uはfp32の1次元配列
 return (1 + np.log(2 * np.pi)) / 2 - ¥
        k1 * (np.mean(np.log(np.cosh(u))) - gamma)**2 - ¥
        k2 * (np.mean(u * np.exp(-u**2) / 2))**2
```

図 6 def_entropy 関数のコード (Python)

```
/* C++言語のソースコード */
<cmath>
using namespace std ;
float a[NUM];
float b[NUM];

/* 配列aには何かしらの入力データが入って
いるものとする */
for(int i=0; i<NUM; i++) {
  b[i] = log(cos(a[i]));
}
```

図 7 高速化対象のコード例 (C++)

対して ∗∗2 を計算し，その各結果に対して-1 をかけて 2 で割り，その各結果に対して exp を計算する．

そこで，配列データに対して算術関数 (log, exp, sin, cos, tan, sinh, cosh, tanh, pow, sqrt) や定数との add/sub/mul,div などの演算を繰り返すような処理を高速化する方法について検討した．以降の節では，説明を簡単にするために図 7 に示すソースを例題にして検討した高速化手法について説明する．

4.2 従来技術

図 7 を処理する場合，従来技術として以下の 2 つの方法が存在する．

(1) math ライブラリに実装されているスカラー関数を使用
(2) math ライブラリに実装されている SIMD 関数を使用

(1) を利用する場合，図 8 に示すフローで処理される．$a[i]$ を入力として cos 関数を call し，$cos(a[i])$ を計算する．次に $cos(a[i])$ で得た値を入力として log 関数を call し，得られた $log(cos(a[i]))$ を $b[i]$ に結果を格納する．1 では，cos 関数と log 関数がそれぞれ NUM 回 call される．

(2) の手法として SLEEF(SIMD Library for Evaluating Elementary Functions)[10][11] を使用する場合を例にして

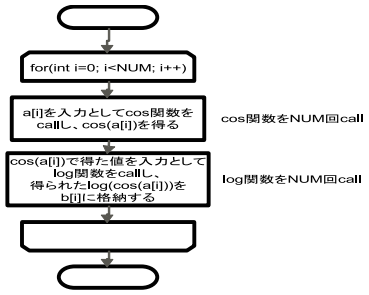

図 8 math ライブラリに実装されているスカラー関数を使用する場合のプログラムの処理フロー

```
/* SLEEFライブラリが提供する関数のプロトタイプ宣言 */
svfloat32_t Sleef_cosfx_u10sve(svfloat32_t a);
/* floatデータ配列を入力としてを受け取り、
  floatデータ配列を返す関数。
  svfloat32_tの要素数はSVEベクトルサイズによって決定される。
  SVEベクトル長が512ビットの場合、要素数は16となる。
  SVEベクトル長が128ビットの場合、要素数は4となる。*/
svfloat32_t Sleef_logfx_u10sve(svfloat32_t a);
```

図 9 SLEEF の提供する関数のプロトタイプ宣言

説明する．SLEEF は，C 標準数学関数をベクタライズ化して実装したライブラリであり，x86, PowerPC64, AArch64[7] 等の様々なプロセッサで利用可能なライブラリである．富岳上で大規模変数の因果探索を行うことを検討しているため，ここでは AArch64 向けの SLEEF について説明する．SLEEF が提供する AArch64 向けの関数のプロトタイプ宣言の例を図 9 に示す．AArch64 は Scalable Vector Extensions (SVE) という拡張命令セットを持っており，ベクトル長が可変であるが，SLEEF は SVE ベクトル長がいくつであってもビルド可能な作りになっている．プログラムが動作する CPU の SVE ベクトル長が 512bit の場合は，svfloat32_t の要素数は 16 となり，16 並列で動作し，CPU の SVE ベクトル長が 128bit の場合は，svfloat32_t の要素数は 4 となり，4 並列で動作する．アーキテクチャが AArch64 で SVE ベクトル長が 512bit の CPU において，AArch64 向けの SLEEF を使用する場合，ソースコードは図 10 のように記述される．SVE ベクトル長が 512bit である場合，svfloat32_t の要素数は 16 となるため，ループの繰り返し回数は要素数に合わせた記述になっている．(2) を利用する場合，図 10 のプログラムは図 11 に示すフローで処理される．Sleef_cosfx_u10sve 関数と Sleef_logfx_u10sve 関数は 1 回の処理で 16 個のデータに対してそれぞれ cos と log の計算を実行できる．したがって，従来技術 2 は従来技術 1 に対して cos 関数と log 関数の呼び出し回数を 16 分の 1 に削減することが可能である．しかし，関数コール回数やメモリアクセス回数に関して更なる高速化が可能であると我々は考えている．次節にて，我々が検討している計算カーネルの高速化手法について説明する．

4.3 JIT 技術を利用した複数関数の処理をフュージョンすることによる計算カーネルの高速化

まず，Just-In-Time(JIT) 技術を利用して実行コードを

```
/* C++言語のソースコード */
#include <sleef.h>

svfloat32_t *s;
svfloat32_t *d;

/* 配列aには何かしらの入力データが入っているものとする */
for(int i=0; i<NUM; i+=16) {
  d = reinterpret_cast<svfloat32_t *>(&(b[i]));
  s = reinterpret_cast<svfloat32_t *>(&(a[i]));
  *d = Sleef_expfx_u10sve(Sleef_cosfx_u10sve(*s));
}
```

図 10 SLEEF 使用時のソースコード

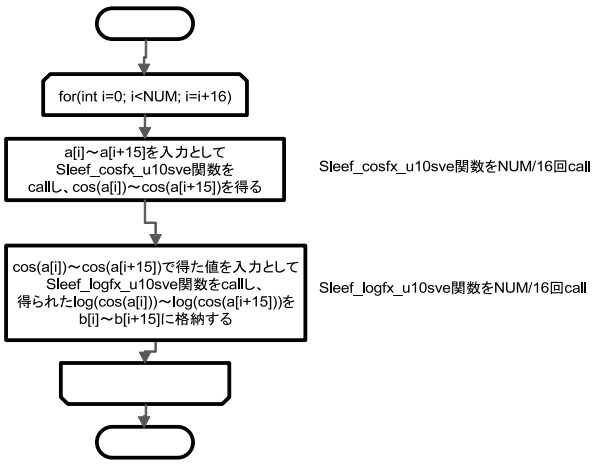

図 11 SLEEF を使用する場合のプログラムの処理フロー

生成することによる高速化について説明する．本手法では，プログラム実行時に決定する情報を利用して，実行時にコード (関数) を生成し，それを用いることで高速に処理を行う．この場合，ソースコードは図 12 のように記述される．gen_v_cos 関数が実行されると，メモリ上に SIMD 命令で cos の計算を行う実行コードが生成され，gen_v_log 関数が実行されると，メモリ上に SIMD 命令で log の計算を行う実行コードが生成され，gen_ret 関数を実行すると，ret 命令の実行コードが生成される．これらの実行コードはメモリ上の連続した領域に生成される．メモリ上に生成した実行コードは gen_exec 関数で実行される．従って，図 12 のプログラムは図 13 に示すフローで処理される．gen_exec 関数は，cos を計算する実行コードと log を計算する実行コードを連続で実行し，終了後に ret 命令を実行する．従来技術 2 では，cos を計算する実行コード終了後に ret 命令が実行され，log を計算する実行コード終了後に ret 命令が実行されていたため，従来技術 2 に比べて ret 命令の実行が NUM/16 回少なくなる．現時点で，従来技術 2 よりも高性能であると考えられるが，生成される実行コードにはまだ改善の余地がある．gen_v_cos 関数と gen_v_log 関数の処理フローを図 14 に示す．生成される実行コードの改善可能な点は次の 4 つである．

(1) 毎回係数データの load を行う
(2) 依存関係のあるレジスタを使う命令が連続する
(3) cos と log の計算を連続する際には、不要な命令が生成される

```
アプリケーションプログラム
gen_v_cos();   /* cos関数の計算をする命令列を生成する */
gen_v_log();   /* log関数の計算をする命令列を生成する */
gen_ret();     /* ret命令を生成する */

/* 配列aには何かしらの入力データが入っているものとする */
for(int i=0; i<NUM; i++) {
  b[i] = gen_exec(a[i]); /* 生成したコードを関数として使用する */
}
```

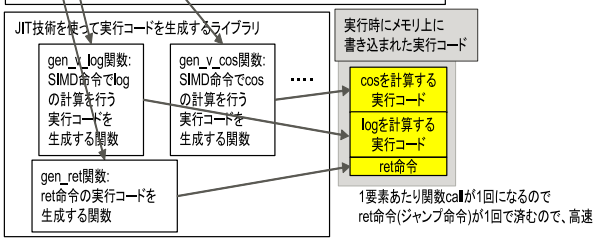

図 12 Just-In-Time(JIT) 技術により実行コードを生成するプログラム

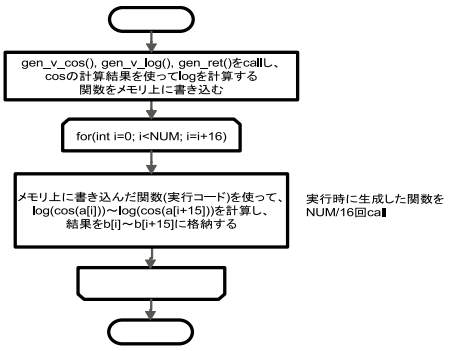

図 13 Just-In-Time(JIT) 技術により実行コードを生成する場合のフロー図

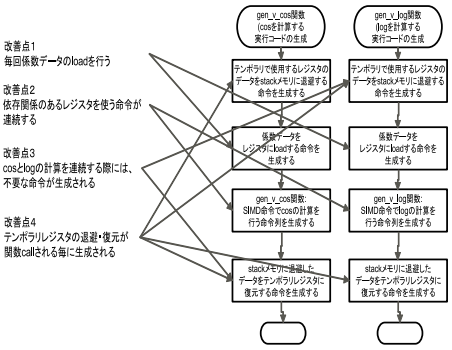

図 14 gen_v_cos 関数と gen_v_log 関数の処理フロー

(4) テンポラリレジスタの退避・復元が関数コールされる毎に生成される

これらの改善可能な点について説明する．SIMD 型の入力データに対して cos を計算する実行コードを生成する例を図 15 に示す．ここでは，説明を簡単にするために，cos 関数は $cos(val) = c0 + c1 * val + c2 * 2^2$ で計算できるものとしている ($c0, c1, c2$ は係数で val は入力データ)．また，計算の途中結果を z1 レジスタ，係数データ $c0, c1, c2$ を z2, z3, z4 レジスタに格納する実行コードを生成することを仮定している．実行コード生成の注意として入力データが SIMD 型で，返り値データが SIMD 型の場合，生成す

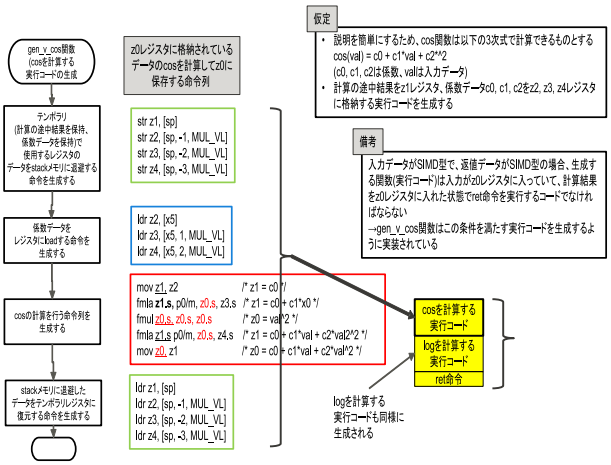

図 15 gen_v_cos 関数による命令列の生成

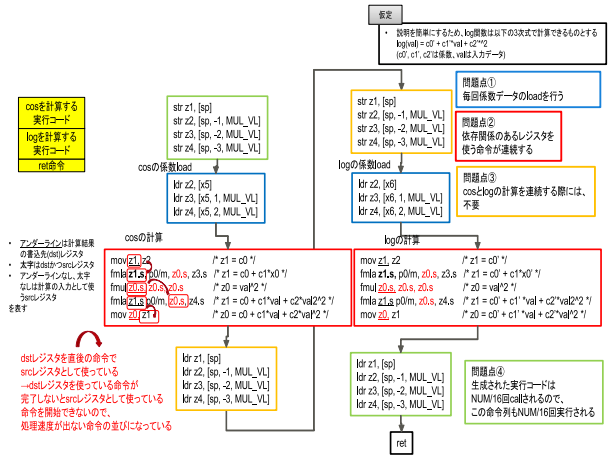

図 16 メモリ上に生成される実行コード

る関数 (実行コード) は入力が z0 レジスタに入っていて，計算結果を z0 レジスタに入れた状態で ret 命令を実行するコードでなければならない．前述の仮定と注意点に従って，gen_v_cos 関数のフローにより生成される命令列は，図 15 の中央の 4 つの四角で示している命令列である．まず，係数データのロード先として使用するレジスタと計算の途中結果を保存するレジスタとして使用するレジスタ (z1, z2, z3, z4) の内容を退避する命令列を生成する (一番上の四角)．次に，計算に必要な係数データをロードする命令列を生成する (上から二番目の四角)．そして，cos を計算する命令列を生成し (上から三番目の四角)，退避したレジスタの内容を復元する命令列を生成する (上から四番目の四角)．

log を計算する実行コードも gen_v_log 関数のフローに応じて同様に生成される．結果として，cos と log を計算する実行コードは図 16 のようになる．

図 16 を用いて，生成される実行コードの 4 つの改善可能な点を説明する．現状の実行コードでは，1 回の関数コールで 16 個の配列データに対して cos と log の計算を行っている．そのため，16 個の配列データに対して処理を行うたびに係数データの load を行う必要がある (改善可能な点 (1))．

また，cos と log を計算する実行コードを見ると，dst レジスタに指定されているレジスタを直後の命令で src レジスタとして使用されている．この場合，dst レジスタを使用している命令が完了しないと src レジスタとして使用している命令を開始できないため，処理速度が出ない命令の並びとなっている (改善可能な点 (2))．

そして，cos を計算する実行コードの後に，テンポラリレジスタの内容を復元・退避を行う命令列が存在するが，cos と log の計算を連続で行う場合，復元したレジスタに対してすぐに退避を実行するだけであり，不要なコードとなっている (改善可能な点 (3))．

さらに，最初と最後に生成されるレジスタの退避・復元を行う命令列は，関数がコールされる度に実行されるコードであるため，全データに対して cos と log の計算を行う場合，NUM/16 回実行されることになる (改善可能な点 (4))．

前述の 4 つの改善可能な点を改善した我々の提案手法である JIT 技術を利用した複数関数の処理をフュージョンすることによる計算カーネルの高速化方法について説明する．本手法におけるプログラムと処理フローを図 17 に示す．本手法の手順は以下である．

(1) あらかじめ関数を処理するための命令列と，関数の処理で必要となる係数の数とテンポラリレジスタの数をテーブルを定義しておく
(2) プログラム上で配列データに対して行う cos や log といった計算を gen_op_add 関数を利用して登録
(3) 登録後，gen_code 関数を利用して配列データに対して登録された計算を順に実行する実行コードを生成
(4) gen_exec を利用して配列の要素それぞれに対し，生成した実行コードを実行

手順 (3) における実行コードの生成フローを図 18 に示す．まず，登録されている計算において必要な係数の個数とテンポラリレジスタの総数を手順 (1) で定義したテーブルを基に算出し，算出した数分のレジスタの内容を退避し，登録された計算に必要なテンポラリレジスタを確保する．

そして，登録されている計算で必要な係数と計算対象の入力データを確保したレジスタにロードする命令を生成する．次に，ロードした計算対象の入力データに対して登録されている順に各計算の命令列を生成し，計算した結果をメモリにストアする命令列を生成する．この時，各計算間において計算結果のストア命令を生成しないため，改善可能な点 (3) を改善している．

また，一度に計算対象の全入力データに対して計算を行うことができない場合を想定して，計算対象の入力データのロードから計算結果をメモリにストアする命令列を計算対象の全入力データに対して繰り返し実行されるようにデータのアドレス計算と jump 命令を生成する．このよう

表 1 従来技術 (1),(2) と提案技術における関数コール回数，係数のロード回数，データのロード・ストア回数の比較結果

項目	従来技術 (1)	従来技術 (2)	提案技術
関数コール回数	$2N$	$N/8$	1
係数のロード回数	$2N$	$N/8$	1
データのロード・ストア回数	$4N$	$4N$	$2N$

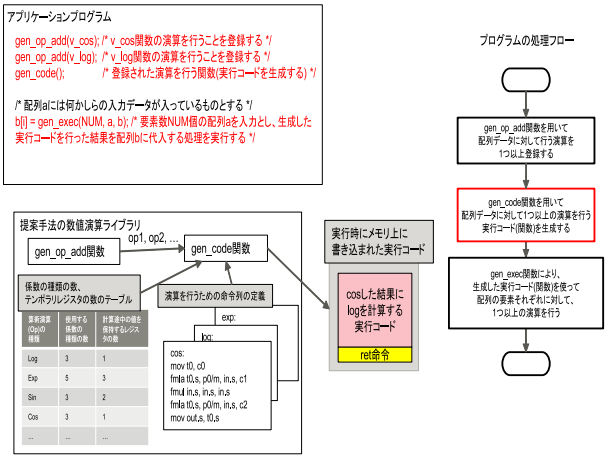

図 17 JIT 技術を利用した複数関数の処理をフュージョンすることによる計算カーネルの高速化手法

にすると，係数データのロードを 1 回行うだけで全入力データに対して計算を行うことが可能になるため，改善可能な点 (1) を改善できる．

さらに，複数の入力データを一度に処理することにより，レジスタの依存関係が発生しないように命令列を生成することができるため，改善可能な点 (2) も改善できる．

最後に，メモリに退避した内容をレジスタに復元する命令を生成し，ret 命令を生成する．本フローにより生成された実行コードを 1 回実行すると全入力データに対して計算を実施できるため，関数コールは 1 回だけ済む．従って，改善可能な点 (4) も改善できる．図 18 により生成される実行コードを図 19 に示す．

関数コール回数，係数のロード回数，データのロード・ストア回数について，観測変数の数を N とした場合の本提案技術と従来技術 (1)，(2) を比較すると，表 1 のようになる．全ての項目について，本提案技術が優れていることが分かる．

一方で，本提案技術には，従来技術には存在しないオーバーヘッドとして実行コードの生成時間が存在する．しかし，実行コードの生成時間を今回のコードを例に計測すると，約 0.15[msec] と非常に小さかった．さらに，実行コードの生成は初回実行時に行われるのみで、次の実行からは生成した実行コードが再利用されるため，Direct-LiNGAM において観測変数の数を N とすると，一度生成した実行コードは，$O(N^3)$ 回再利用されることになる．従って，観測変数の数が 1 万以上と十分に大きい場合，実行コードの生成時間は無視できる大きさであるため問題ないと言える．

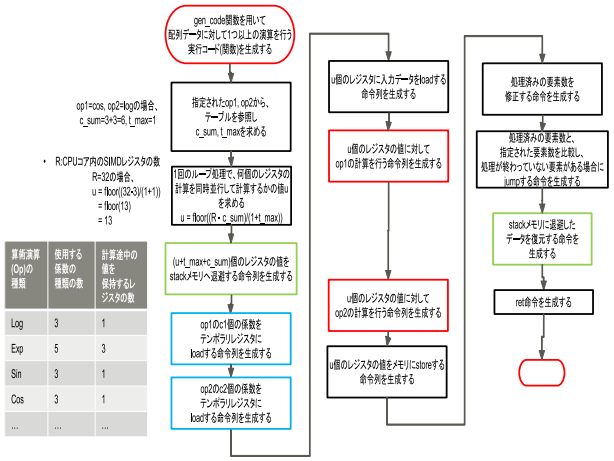

図 18 gen_code 関数による実行コードの生成フロー

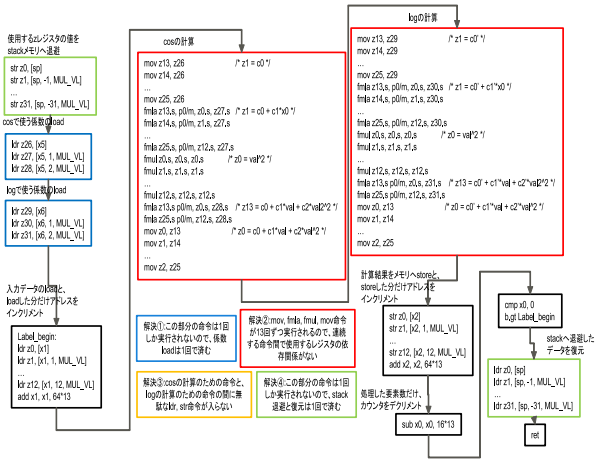

図 19 gen_code 関数による実行コードの生成フロー

5. 評価

従来技術と 4.3 で説明した提案技術それぞれで処理した場合の実行時間を測定した．評価対象の疑似コードを図 20 に示す．今回，配列データに対する演算を $exp(exp(a[i]))$ としているのは，exp の JIT 実装が既に済んでおり，すぐに評価に流用できたためである．図 6 で使用されている log や $cosh$ に関してはこれから実装を行う予定である．評価環境には，スーパーコンピューター富岳上で大規模変数の因果探索を行うことを検討しているため，AArch64 アーキテクチャの CPU である A64FX[8][9] が搭載されている FX700 を使用した．A64FX の SVE ベクトル長は 512bit である．A64FX には複数の CPU コアが搭載されているが，本評価では 1 コアのみで動作させた場合の評価となっている．

従来技術 1 としては cmath ライブラリを使用し，従来技術 2 としては 4.2 で説明した SLEEF を使用した．評価環境として使用した A64FX の SVE ベクトル長が 512bit であるため，SLEEF は 16 並列で動作する．提案技術の命令列の生成は，AArch64 向け JIT アセンブラである

```
/* C++言語のソースコード */
<cmath>
using namespace std ;
float a[NUM];
float b[NUM];

/* 配列aには何かしらの入力データが入って
いるものとする */
for(int i=0; i<NUM; i++) {
  b[i] = exp(exp(a[i]));
}
```

図 20 評価対象の疑似コード

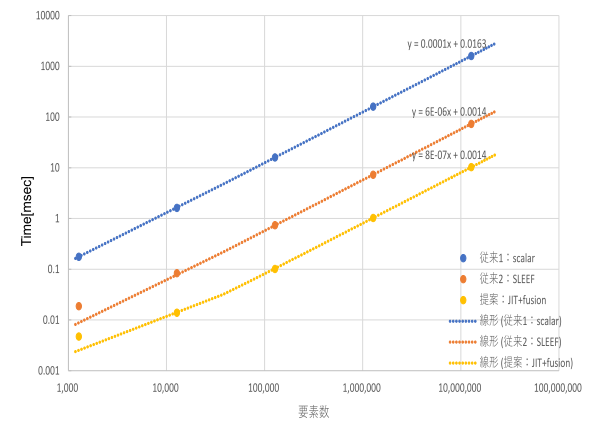

図 21 従来手法と提案手法の評価結果

Xbyak_aarch64[12][13][14] を使用した．Xbyak_aarch64 では AArch64 命令のニーモニックを名前とする関数群が実装されており，これらを用いて所望の機械語列が生成されるように実装する．Xbyak_aarch64 は SVE 拡張を含む AArch64 アーキテクチャで実行可能な機械語列の生成に対応しており，標準的な C++コンパイラでビルドでき，C++で実装するプログラムから使用可能である．

従来技術 1，従来技術 2 と提案技術の測定結果を図 21 に示す．グラフの横軸が配列の要素数を示し，グラフの縦軸が実行時間 [msec] を示す．提案技術を利用することにより，従来技術 1 に比べて約 150 倍，従来技術 2 に比べて約 6 倍の性能改善が確認できた．

6. まとめ

OSS として公開されている現状の Direct-LiNGAM を利用して，1 万変数以上の大規模変数に対して因果探索を行うことは現実的には難しい．そこで，筆者らは Direct-LiNGAM に対して 3 つの高速化を検討しており，本稿では，検討中の 3 つの高速化のうちの一つであるシングルスレッドあたりの処理の高速化として，計算カーネルの高速化に関して検討し，性能評価を行った．結果として，我々が提案する JIT 技術を利用した複数関数の処理をフュージョンすることによる計算カーネルの高速化手法により、従来技術 1 に対しては約 150 倍の性能向上を，従来技術 2 に対しては約 6 倍の性能向上を期待できることが分かった．

今後は，残りの高速化施策であるループの並列処理化の検討を進めることにより，Direct-LiNGAM による 1 万変数以上の大規模変数に対しての因果探索の実現を目指す．

参考文献

[1] S. Shimizu, P. T. Hoyer, A. Hyvrinen and A. Kerminen: A Linear Non-Gaussian Acyclic Model for Causal Discovery, Journal of Machine Learning Research, 7(Jul): 2003–2030, 2006.
[2] S. Shimizu, T. Inazumi, Y. Sogawa, A. Hyvrinen, Y. Kawahara, T. Washio, P. O. Hoyer and K. Bollen: DirectLiNGAM: A direct method for learning a linear non-Gaussian structural equation model, Journal of Machine Learning Research, 12(Apr): 1225–1248, 2011.
[3] A. Hyvrinen and S. M. Smith: Pairwise likelihood ratios for estimation of non-Gaussian structural eauation models, Journal of Machine Learning Research 14:111-152, 2013.
[4] 清水昌平：統計的因果探索（機械学習プロフェッショナルシリーズ），講談社 (2017).
[5] LiNGAM(online), 入手先 ⟨https://github.com/cdt15/lingam⟩ (accessed 2021-05-14).
[6] Fujitsu and RIKEN Complete Joint Development of Japan ' s Fugaku, the World ' s Fastest Supercomputer, https://www.fujitsu.com/global/about/resources/news/press-releases/2021/0309-02.html (accessed 2021-06-16).
[7] Arm Limited or its affiliates, Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile, 2021.
[8] Toshio Yoshida: "Fujitsu High Performance CPU for the Post-K Computer," in Proc. Hot Chips 30, Aug. 2018
[9] A64FX(online), 入手先 ⟨https://github.com/fujitsu/A64FX⟩ (accessed 2021-06-07).
[10] Naoki Shibata and Francesco Petrogalli: SLEEF: A Portable Vectorized Library of C Standard Mathematical Functions, in IEEE Transactions on Parallel and Distributed Systems, DOI:10.1109/TPDS.2019.2960333 (Dec. 2019).
[11] SLEEF(online), 入手先 ⟨https://github.com/shibatch/sleef⟩ (accessed 2021-06-17).
[12] K. Kawakami, S. Moriyuki, K. Kurihara and N. Fukumoto: Xbyak_aarch64;JIT Assembler for Next Generation Supercomputer, *Proc. CoolCHIPS23*, (online), (2020).
[13] K. Kawakami: Xbyak_aarch64; Just-In-Time Assembler for Armv8-A and Scalable Vector Extention, https://connect.linaro.org/resources/lvc21/lvc21-203/ (accessed 2021-06-15), Linaro Virtual Connect 2021, (online), (2021).
[14] Xbyak_aarch64(online), 入手先 ⟨https://github.com/fujitsu/xbyak_aarch64⟩ (accessed 2021-06-14).