Vol.2021-HPC-180 No.3
2021/7/20

統一的なオープンソース線形代数ライブラリ monolishの提案

菱沼 利彰[1]　五十嵐 亮[1]　寺村 俊紀[1]　森田 直樹[1,2]　井原 遊[1]

概要：近年，プロセッサアーキテクチャの多様化が進んでいる．数値シミュレーションを様々なアーキテクチャで行うことを考えたとき，CPU では BLAS や LAPACK を用いれば密行列に対する演算が統一された API で利用できるが，アクセラレータでは多くの場合データ通信の制御や API の変更が必要になる．疎行列に対する演算は，一部のハードウェアベンダ製ライブラリが疎行列に対する演算を備えているが，全てのアーキテクチャで動作はせず，統一された API は定義されていない．著者らは，真に有用な線形代数ライブラリは実用されているすべてのアーキテクチャで動く必要があるとの考えから，各ベンダやライブラリのデータ型，行列格納形式，データ通信 API などを扱うための統一された API を定義し，すべてのアーキテクチャで動作する可搬性の高いオープンソースライブラリ monolish を提案する．本研究ではこの第一段階として，シングルノード，シングルデバイス向けに開発した提案ライブラリの性能を評価した．行列行列積，LU 分解，共役勾配法のプログラムを 8 つの環境で実行し，提案したライブラリを用いることですべての環境でプログラムを変更せずに動作する高性能かつ可搬性の高いプログラムを実現できることを示した．

1. はじめに

プロセッサアーキテクチャの多様化が進んでいる．CPU では Intel, AMD の x86_64, IBM の Power に加え ARM の ARM64, 特に富士通 A64FX [1] の期待が高まっている．アクセラレータでは NVIDIA や AMD の GPU に加え，NEC から SX-Aurora TSUBASA (SXAT) [2] が登場した．

これらのプロセッサアーキテクチャは性能を出しやすいアプリケーションが異なっている．そのためユーザがアプリケーションの性能，ジョブの混雑状況，利用料金などから自由に計算機を選択できれば，ユーザ側，システム運用側の双方にとって効率がよい．

一方，数値シミュレーションを様々なアーキテクチャで行うことを考えると，プロセッサアーキテクチャや動作する数値計算ライブラリの違いから，プログラムの大幅な修正が必要になることや，性能が出ないことが問題になる．

数値シミュレーションの核である連立一次方程式や固有値の求解アルゴリズムは，ベクトル，密行列，疎行列に対する演算によって構成される．ベクトルや密行列に対する演算は，CPU では BLAS [3] や LAPACK [4] を用いれば統一された API で利用できる．一方，アクセラレータでは多くの場合データ通信の制御 API の利用や，デバイスメモリの管理が必要になる．

BLAS や LAPACK の関数のインタフェースにも課題がある．例えば BLAS では単精度内積の関数を “sdot”，倍精度内積の関数を “ddot” と定義している．そのため倍精度で動くプログラムを単精度に変更する場合，すべての関数呼び出しを変更する必要がある．また，BLAS には単精度，倍精度以外の半精度や 4 倍精度など向けの関数は定義されていないため，これらに対応するためには独自のインタフェースを新たに定義する必要がある．

疎行列に対する演算は，一部のハードウェアベンダ製の数値計算ライブラリ (ベンダ製数値計算ライブラリと呼ぶ) が備えているが，全てのアーキテクチャで動作せず，統一された API は定義されていない．そのため疎行列に対する演算においてライブラリを使用することは可搬性の低下に繋がる．また，高速な疎行列計算のためには，CRS や ELL [5] などの非零要素のみを格納した格納形式を行列やアーキテクチャに合わせて選択する必要がある．

このようなプロセッサアーキテクチャごとのソフトウェア環境の違いはアルゴリズムを実装するユーザにとって負担であると考えられ，ハードウェアに最適化された線形代数の実装と，それらを利用するシミュレーションアルゴリズムの実装を明確に切り分けることが必要である．

著者らは，真に有用な線形代数ライブラリは実用されて

[1] 株式会社科学計算総合研究所
[2] 筑波大学

ⓒ 2021 Information Processing Society of Japan

いるすべてのアーキテクチャで動く必要があるとの考えから，各ベンダやライブラリのデータ型，行列格納形式，データ通信 API などを扱うための統一された API を定義し，ユーザは 1 つのプログラムを作成するだけで，すべてのアーキテクチャで動作する可搬性の高いオープンソースの線形代数ライブラリを開発することを提案する．

ライブラリを monolish (MONOlithic LIner equation Solvers for Highly-parallel architecture) と命名し，monolish によって統一したい課題として，以下の 4 つを定めた．

課題 1　データ型に依存した関数のインタフェース

課題 2　ホスト，デバイス間のデータ転送 API やメモリ管理

課題 3　数値計算ライブラリのインタフェース

課題 4　行列の構造や格納形式ごとの関数のインタフェース

すべてのプログラムを共通化した場合，性能が出ないことが問題になると考えられる．そのため可能な限り性能の低下を起こさないために実装すべき機能を定義すること，性能への影響の少ない実装の共通化を行うこと，可能な限り高速なベンダ製数値計算ライブラリを呼び分けることが重要になる．

本論文では，その第一段階としてシングルノード，シングルデバイスの環境を仮定し，いくつかの計算を対象として，様々な計算機システムでプログラムの変更せずに動作させるために必要な機能やインタフェースについて論じる．様々な計算機システムでプログラムを変更せずに動作させることで monolish の可搬性を示し，その性能について評価する．2 章でそれぞれの課題を解決するためのライブラリの設計について述べ，3 章で monolish 内部で様々なアーキテクチャ向けにどのようなプログラムを実装するかについて述べ，4 章で monolish を用いて実装したプログラムをいくつかのスーパーコンピュータシステム上で実行した際の性能を評価する．

2.　monolish の設計

本節ではそれぞれの課題を解決するための monolish の設計について論じる．また，この設計に基づいて開発した monolish を用いて実装された共役勾配法のプログラムを付録 **図 A·1** に載せた．

2.1　課題 1 データ型に依存した関数のインタフェース

問題やアーキテクチャによって計算精度を変更することを考えたとき，BLAS，LAPACK のような C, Fortran から使われることを前提とした "型名+演算名" の関数の命名規則によるインタフェースは，変数の精度を変更するごとにプログラム中の全ての関数呼び出しを変更しなければならないため現実的ではない．

SLATE プロジェクト [6] に含まれる BLAS++，LAPACK++では，オブジェクト指向型の言語である C++のテンプレートや関数のオーバロード機能を用いて BLAS, LAPACK に対して型に依存しないインタフェースを提供し，性能劣化なく高い性能が提供できることを示しているが，アクセラレータ向けのプログラムへの対応はまだ行われていない [7]．

著者らは BLAS++, LAPACK++を参考に，アクセラレータにも対応した型に依存しない BLAS，LAPACK インタフェースを実装した．

2.2　課題 2 ホスト，デバイス間のデータ転送 API やメモリ管理

はじめに，ユーザが最も多いと考えられる Intel CPU と NVIDIA GPU について考えた．1 つのプログラムで CPU と GPU の両方で動作するようにする．

CPU と GPU のプログラムの大きな違いは，ホストとデバイスの通信を記述しなければならないこと，ホストメモリとデバイスメモリを管理するためのポインタ型変数を両方管理しなければならないことである．

はじめに通信の記述について考える．一般的に CPU とくらべてアクセラレータのメモリは小さいこと，性能のためには通信は最小限にする必要があることから，デバイスへのデータの通信を自動的に行うことは難しい．そこで，CPU 向けに記述したプログラムを GPU で動かすのではなく，GPU 向けに記述したプログラムを CPU でも動くようにすることを考えた．これは，データ転送用の関数の内部で通信を行うか，何も行わないかを切り替えるだけでよい．

データ転送用の関数の挙動をユーザが指定するための方法として，共有ライブラリを複数作成する方法を選んだ．内部ではマクロを用いて実装を切り替え，ライブラリのビルド時に指定して切り替える．GPU 向けの共有ライブラリはデータ転送の関数は GPU にデータを転送するが，CPU 向けの共有ライブラリは何も行わない．これによりユーザは動かしたい環境によってリンクするライブラリを切り替えるだけでよい．

将来的にマルチデバイスに対応することを考えた場合，1 プロセスに複数のデバイスを割り当てることはできなくなるが，1 プロセスに 1 デバイスを対応付ければ，このインタフェースで問題なく CPU と GPU を切り替えられると考えている．

次にデバイスメモリを管理するためのポインタ型変数の扱いについて考える．CUDA [8] では，デバイスメモリ用のメモリアロケート関数が用意されており，通常の C の malloc を用いた場合はホストメモリ，CUDA のメモリアロケート関数を用いた場合はデバイスメモリが確保される．このような実装の場合，CPU と GPU の 2 種類の変数を管

理しないといけないため，コードの共通化ができない．

そこで，ベクトルや行列を扱うクラスを作成し，状態管理をクラスに行わせることで，ユーザからは 1 つの変数として扱えるようにする．データがホストメモリとデバイスメモリのどちらを参照したいのかをフラグで管理し，ユーザにデータ転送用の関数を実行させることでフラグの状態を切り替える．フラグの状態によって各関数内の挙動を切り替える．

このようなフラグ管理にするため，GPU が有効な状態で CPU のデータを参照すること，CPU と GPU を並列に計算することは考えない．また，フラグの状態が曖昧になるデバイスへの非同期通信も行わない．非同期通信を行わない場合，ユーザはホスト，デバイス間のデータフローだけを記述すればよい．

ホスト，デバイス間のデータフローの制御に必要な関数として以下の 4 つを考えた．

send デバイスメモリの確保とデータ送信，フラグ有効化
recv データ受信とデバイスメモリの解放，フラグ無効化
nonfree_recv データ受信
device_free メモリの解放，フラグ無効化

複数の send を行った場合はメモリの確保が重複してしまうが，フラグを参照して send 内で切り替えればよく，update のような関数は必要ない．内部の実装の共通化の方法については，3 章で詳しく述べる．

2.3 課題 3 数値計算ライブラリのインタフェース

CPU のコア数は増加している．例えば SQUID [9] のフロントエンドノードでは 38 コアの CPU4 基が搭載されており，合計で 152 コアが使用できる．一般的には，行列に対する演算はベクトルに対する演算と比べて相対的に多くの時間がかかると考えられてきた．しかし，一般的に疎行列の非零要素が多くても 100 程度であることを考えると，疎行列ベクトル積が 100 コア以上の CPU で十分に並列化された場合，並列化されていないベクトル演算と計算時間が同程度になってしまうことも考えられる．このことから，並列化されていない関数は可能な限り提供すべきではない．

このことから，ベクトルや行列に対する操作，演算を定義し，可能な限り並列化された関数を提供する．すべてを実装することは難しいことから，著者らはベンダ製数値計算ライブラリが並列化されていて最も高速だという仮定をおき，可能な限りこれらを呼び出し，実装されていないもののみを monolish に実装することとした．

はじめに，各ライブラリの実装状況を調査した．対象は Intel MKL (MKL と呼ぶ, バージョンは 2020.0166), CUDA の cuBLAS, cuSPARSE, cuSOLVER などのライブラリ (これらを総称して CUDA ライブラリと呼ぶ，バージョンは CUDA10.1)，NEC NLC, FUJITSU SSL II とした．

単純にヘッダに含まれる関数の数を数えた場合，BLAS は約 150 関数，LAPACK は約 1,500 の関数が実装されている．一方，ベンダ製数値計算ライブラリの中で最も機能数が多いのは MKL で，10,000 関数以上が実装されている．また，最も機能が少ないのは CUDA ライブラリで合計で約 1,000 関数が実装されている．

このようにアーキテクチャごとに利用できる数値計算ライブラリの関数に大きな違いがある．最も機能の多い MKL を参考に，実装すべき関数を以下の 5 つのグループに分けて考えた．

- BLAS: ベクトル，密行列に対する基本演算
- VML: 配列に対する数学関数
- Dense Solver: 密行列に対する線形ソルバ
- Sparse BLAS: 疎行列に対する基本演算
- Sparse solver: 疎行列に対する線形ソルバ

疎行列に対する演算において，インタフェースが同じソフトウェアは 1 つもなかった．また，対応している疎行列の格納形式にも差があり，すべてのアーキテクチャの疎行列演算ライブラリをサポートし続けることは難しいことがわかった．

そのため最初の実装として，最もユーザが多いと考えられる MKL, CUDA ライブラリの呼び出しを実装し，それ以外は monolish の実装した汎用プログラムに切り替えることにした．MKL のを使用する場合を “Intel, ” CUDA ライブラリを使用する場合を “NVIDIA”, それ以外を “OSS” と呼ぶ．これらの切り替えで使用されるライブラリを**表 1** に示す．将来的には，これら以外の切り替えも実装することで，より多くのプロセッサアーキテクチャに対応できると考えている．

CUDA ライブラリ以外では，共通している関数は BLAS に含まれる演算のみであった．そのため BLAS グループの実装は CPU かアクセラレータで切り替えればよく，あとはリンク時に適切な BLAS をリンクすればよい．

BLAS には，LAPACK の実装に必要な axpy ($y = ax+y$) や scal ($y = ax$) が実装されているが，ベクトル和などは実装されていない．この問題を解決するため，MKL では BLAS, LAPACK に加えて Vector Math Library [10] と呼ばれる配列の各要素に対する数学関数ライブラリが実装されている．これは共役勾配法などに含まれる $r_0 = b - Ax$ などの計算で，ベクトルの減算を行うときに役立つと考え，“VML” グループと呼んで実装することにした．VML は MKL にしか存在しないため，その他のアーキテクチャでは全て “OSS” が使用される．

“Dense Solver” グループには LU 分解，QR 分解，Cholesky 分解などが含まれる．CUDA ライブラリには cuBLAS に BLAS のほぼすべての関数，cuSPARSE に疎行列に対する演算が実装されているものの，cuSOLVER は LAPACK のすべての機能を持たない．特に，一般行列

表 1　各実装の切り替えで使用されるライブラリ

	Intel	NVIDIA	OSS
BLAS	MKL	cuBLAS	CBLAS 互換
VML	MKL	monolish	monolish
Dense Solver	MKL	cuSOLVER	LAPACK 互換
Sparse BLAS	MKL	cuSPARSE	monolish
Sparse Solver	monolish	monolish	monolish

(GE) 以外の関数はほとんど実装されていない．そのため，LAPACK の密行列以外の形式は対象外とすることにし，CUDA ライブラリと LAPACK に共通した機能のみを実装することにした．

“Sparse Solver” グループには Jacobi 法や共役勾配法 (CG 法) [11] などが含まれる．MKL には共役勾配法の関数が提供されているが，これらは一般的に “BLAS” グループ，“Sparse BLAS” グループ，“VML” グループに含まれる関数の組み合わせで実装できる．各ライブラリに含まれる関数の数やインタフェースに差があることから，これらはライブラリを使用せず，monolish に独自に実装する．

“OSS” に必要なのは CBLAS, LAPACK 互換のインタフェースをもつライブラリのみであるので，将来のアーキテクチャにおいても動作することは十分に期待できると考えている．

2.4 課題 4 行列の構造や格納形式ごとの関数のインタフェース

python の Scipy [12] には, COO, CRS, CSC, DIA, BCRS [5] など，様々な疎行列フォーマットのクラスが実装されており，すべてのクラスに対して同一のメンバ関数を実装することで，同じように扱えるようにしている．

しかし，Scipy のこのようなアプローチは性能面で問題がある．例えば COO 形式の疎行列は，行列の要素の追加や参照は簡単に行えるが，疎行列ベクトル積は並列化が難しく，他の格納形式と比べて性能が出ない．一方で CRS 形式の疎行列は，疎行列ベクトル積を並列に実行でき，COO より高い性能が期待できるが，行列の要素の追加や参照の性能は出しにくい．これらの違いは行列の格納形式や並列計算の知識がないと判別できないことであり，ユーザが使い分けることは困難であると考えられる．

C++で記述された monolish にユーザが期待することは，性能と利便性の両立であると考えられる．そのため遅い機能はあえて実装しないことで，ユーザのプログラムの性能を保証することを考えた．

各格納形式の実装に差がある場合，格納形式を切り替えるたびにプログラム全体を書き換える必要が生じる．どのクラスに何の機能が実装されているかをわかりやすくするため，疎行列クラスと関数に対して “editable” と “computable” という 2 つの属性を付与して考えることにした．“editable” は行列要素への参照や編集ができることを意味し，“computable” は並列化された行列計算や，アクセラレータでの計算ができることを意味する．

また，疎行列と密行列のどちらを使うべきかは非零要素の分布や数によって異なるが，その切替基準はアーキテクチャによって異なる．そこで，monolish では密行列も疎行列の 1 つとして扱い，密行列のクラスとして Dense, COO 形式のクラスとして COO, CRS 形式のクラスとして CRS を実装した．また，これに伴って関数名も BLAS 準拠の GEMM や GEMV ではなく matmul や matvec とした．

COO には “editable”, CRS には “computable”, Dense には “editable” と “computable” の両方の属性を付与した．これにより，ユーザが行列の作成に使用できるのは COO と Dense のみ，行列計算に使用できるのは Dense と CRS のみとなる．変換の処理は必要であるが，並列化されていないプログラムを誤って実行してしまうことをなくせる．

これにより，同属性をもつ格納形式であれば，プログラム全体を変更せずに行列の宣言部のみを変更するだけで格納形式を変更することができる．例えば，付録の図 A·1 は，“computable” 属性をもつ Dense, CRS の両方で動作することを確認している．将来的には属性をさらに増やす必要がある可能性はあるが，このように属性を分けて実装することで，高速な実装に対し統一した API を付与することができる．

3. 様々なプロセッサで動作するプログラムの実装

本章では，monolish 内部の関数を実装するための様々なプロセッサで動作するプログラムの実装方法を述べる．

複数のアーキテクチャをサポートすることを考えると，特殊なコンパイラやツールを必要とすることは好ましくない．そこで著者らは実装の基盤として LLVM ベースのコンパイラを選んだ．LLVM プロジェクトに含まれる Clang は標準的な C/C++コンパイラであり，Clang での動作を保証することで，他のコンパイラでも十分に動作することが期待できる．

一般的に，アクセラレータは CPU と違い独自のプログラミングモデルを用いることが多い．例えば，NVIDIA GPU では CUDA を使用するのが一般的だが，AMD GPU では CUDA は動作せず，代わりに HIP [13] と呼ばれるプログラミングモデルが提供されており，CUDA とはインタフェースが異なる．今後，新しいアクセラレータが登場する可能性を考えると，個別に実装することは好ましくない．

そこで，2.2 節で定義した通信の API について共通化を考える．アクセラレータに対するプログラムを統一されたディレクティブで記述することができるコンパイラ拡張を利用し，通信 API を共通化する．

Clang から利用できる GPU 向けのコンパイラ拡張として OpenACC と OpenMP Offloading [14] がある．これら

はどちらも Clang において libomp という同じライブラリを利用しており，最終的な性能は libomp の実装に起因すると考えられる．また，これらに実装されている機能に大きな差はない．CPU 向けのプログラムは OpenMP で並列化することから，OpenMP Offloading を用いて GPU 向けのプログラムを実装することにした．

OpenMP Offloading は GPU 等のアクセラレータを OpenMP の指示句を用いて制御するためのコンパイラ拡張であり，OpenMP 4.0 から使用できるようになった．Clang 11.0.1 において，OpenMP Offloading は NVIDIA GPU, AMD GPU の両方をサポートしている [15].

多くの関数で CUDA ライブラリを呼び出す際, OpenMP Offloading によるオーバヘッドがどの程度あるのかは不明である．そこで，国立研究開発法人産業技術総合研究所の AI 橋渡しクラウド (AI Bridging Cloud Infrastructure, ABCI) [16] に搭載されている NVIDIA Tesla V100 上で Clang 11.0.1 に実装されている OpenMP Offloading の性能を検証した．

図 1 に，OpenMP Offloading によるデータ転送のプログラムを，**図 2** に OpenMP Offloading による axpy ($y = \alpha \times x + y$) のプログラムの例を示す．

図 1 は，GPU へデータを送信する関数 send と受信する関数 recv である．send された関数は先頭ポインタが変わらない限り，“pragma omp target” 句の中で GPU 上のメモリとして扱われる．変数名の管理は OpenMP によって自動的に行われるため，デバイスメモリのための変数を用意する必要はない．

図 2 に示す OpenMP_cuBLAS_axpy 関数は，OpenMP Offloading 句を用いて cuBLAS を呼び出す関数である．OpenMP_axpy 関数は，データは事前に GPU に転送済のものとし，for ループを GPU で実行する．この 2 つの関数を，CUDA から cuBLAS を呼び出す関数の性能と比較し，OpenMP Offloading のオーバヘッドを評価する．

図 3 に axpy におけるベクトルサイズ N を 10^3 から 10^8 まで変化させたときの CUDA から cuBLAS を実行した性能を 1 としたときの OpenMP_cuBLAS_axpy 関数と OpenMP_axpy 関数の相対性能を示す．結果には 1,000 回試行した際の平均時間を用い，通信時間は含めていない．

OpenMP_cuBLAS_axpy 関数の性能は，ベクトルサイズが小さい $N = 10^5$ までは CUDA から cuBLAS を呼び出した場合と比べて約 10%性能が高いが，ベクトルサイズが十分に大きい $N = 10^6$ 以上では，性能差は 1%以内に収まった．ベクトルサイズが小さいときの性能が高い理由は不明であるが，OpenMP Offloading のオーバヘッドは小さく，CUDA を記述する場合と比べてほぼ同様の性能が出せることがわかった．

一方，OpenMP_axpy 関数の性能は，ベクトルサイズが小さい $N = 10^5$ までは CUDA から cuBLAS を呼び出した場合と比べて約 20%から 30%性能が高いが，ベクトルサイズの増加に従って性能が低下し，$N = 10^8$ では CUDA から cuBLAS を呼び出した場合と比べて約 10%性能が低い．サイズが小さいことから OpenMP Offloading を用いたことによるカーネル呼び出しのオーバヘッドなどではなく，カーネルの最適化が問題であると考えられる．

これらの結果から，CUDA ライブラリを呼び出す上では，OpenMP Offloading によるオーバヘッドは小さく，CUDA から呼び出す場合とほぼ同等の性能が期待できるため，非常に有用である．OpenMP_axpy 関数のように自分で関数を実装する場合は cuBLAS ほどの性能は期待できないが，cuBLAS と比べて 10%程度の性能差しかないということは良い結果だと考えられる．行列行列積のような計算量が多く，高度な最適化を必要とする関数の実装には向かないが，“VML” などを実装する上では，十分に有用であると考えられる．これらのことから monolish では，アクセラレータ向けのプログラムを OpenMP Offloading を用いて開発することにした．

4. 実験

本章では，monolish を利用するプログラムをいくつか作成し，それぞれリンクするライブラリを切り替えていくつかのスーパーコンピュータシステムで実行し，その性能について評価する．密行列に対する計算として行列行列積と LU 分解，疎行列に対する計算としてベクトル演算，行列ベクトル積から構成される CG 法を対象とした．

4.1 実験環境

Intel CPU, NVIDIA GPU が使用できる環境として ABCI, Fujistu A64FX が使用できる環境として，FOCUS スパコン X システム [17]，Intel CPU, AMD CPU，および SXAT が使用できる環境として大阪大学の SQUID を用いた．それぞれのシステムに対して本論文で用いる名称とハードウェア構成を**表 2** に示す．

ここでアクセラレータは A_NVIDIA と S_SXAT であるが，SXAT にはホスト CPU で動作する VEOS と呼ばれるベクトル型アクセラレータボード上で動作するプロセスを制御するソフトウェアが提供されている．VEOS は VE からは OS のように見えており，VE に Linux システムコールなどを提供する．I/O やシステムコールなどの処理は自動的に VEOS を通じてホスト CPU と協調して行われる．o そのため，EC コンパイラを用いて VE 向けにコンパイルしたプログラムを VH から実行するだけで，VEOS によってプロセスが VE 上に展開されて処理が行われ，ユーザがプログラムを変更したり，VE と VH 間の転送を意識する必要はない [18]．そのため本論文では SXAT も CPU として扱う．

Intel CPU では，monolish は “Intel” の MKL と “OSS”

```
void send(double* data, int N){
    #pragma omp target \
        enter data map(to : data [0:N])
}

void recv(double* data, int N){
    #pragma omp target \
        exit data map(from : data [0:N])
}
```

図 1　OpenMP Offloading によるデバイスへのデータ転送

```
void OpenMP_cuBLAS_axpy(double alpha, double*
    x, double* y, int N) {
#pragma omp target data use_device_ptr(x, y)
    {
        cublasHandle_t h;
        cublasCreate(&h));
        cublasDaxpy(h, N, &alpha, x, 1, y, 1)
            );
        cublasDestroy(h);
    }
}

void OpenMP_axpy(double alpha, double* x,
    double* y, int N) {
#pragma omp target teams distribute parallel
    for
    for (int i = 0; i < N; i++) {
      y[i] = alpha * x[i] + y[i];
    }
}
```

図 2　OpenMP Offloading によるデバイスでの axpy

の OpenBLAS [19] を利用できる．そこで，Intel CPU ではそれぞれ性能評価を行うこととして，合計 8 つの構成に対し性能評価を行うこととした．それぞれのソフトウェア環境を **表 3** に示す．ここで，ソフトウェア名を “monolish” としているものは，monolish が独自に実装した関数を呼び出しているという意味である．

コンパイルオプションとして，“-O3 -fopenmp”，および適切な BLAS や LAPACK をリンクするためのオプションをつけた．これに加え，A64FX にはバックエンドを Clang にするオプション “-Nclang” と自動ベクトル化のためのオプション “-fvectorize” をつけ，大島らの研究 [20] を参考に環境変数として “OMP_STACKSIZE=8m”, “XOS_MMM_L_PAGING_POLICY=demand:demand:demand” を指定した．GPU では NVIDIA Tesla V100 に対して

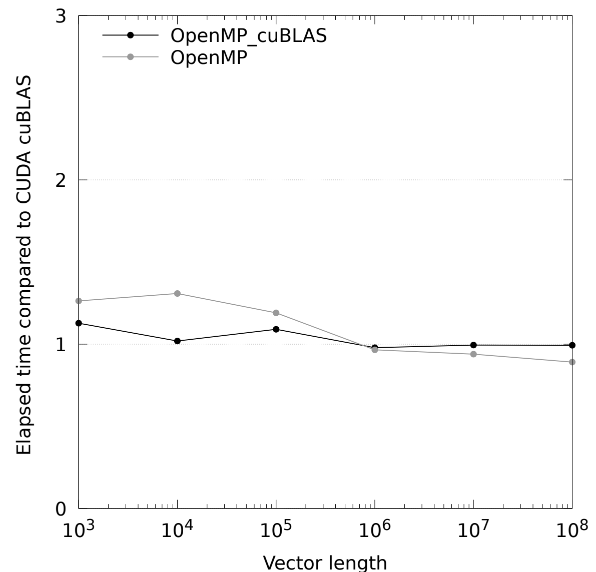

図 3　CUDA から cuBLAS を実行した場合を基準とした OpenMP Offloading によって実装された axpy の相対性能，“OpenMP_cuBLAS_axpy” は OpenMP の Offloaing 句内で cuBLAS を実行．“OpenMP_axpy” は cuBLAS を使わずにループを OpenMP Offloading を用いてデバイスで実行．

OpenMP Offloading を有効にするためのオプションとして “-fopenmp-targets=nvptx64 -Xopenmp-target -march=sm_70” をつけた．

4.2 行列行列積の性能

本節では，$N \times N$ の正方乱数密行列 A, B, C に対する行列行列積の性能を評価する．

作成したプログラムに対しコンパイラとリンクするライブラリを変更するだけで，すべての環境においてプログラムを変更することなく実行できることを確認した．

各アーキテクチャにおける行列行列積の性能を評価した．単精度行列行列積 (SGEMM) の結果を**表 4**, 倍精度行列行列積 (DGEMM) の結果を**表 5** に示す．性能は 30 回試行した平均時間から計算量を $2N^3$ として求めた．このとき，A_NVIDIA では行列はあらかじめ GPU に転送済とし，GPU との通信時間は評価に含んでいない．

実験の結果から，ベンダ製数値計算ライブラリを使用した場合，最も大きい行列サイズ $N = 5,000$ ではすべてのシステムにおいて 65%以上のピーク性能比を発揮した．最もピーク性能比が高いのは S_SXAT における $N = 5,000$ の DGEMM で，これはピーク性能の 95.9%であることから性能は十分高いものと考えられる．このことから，C++ を用いて BLAS の型依存を隠蔽し，内部でベンダ製数値計算ライブラリを呼び分けることのオーバヘッドは小さく，本研究の提案によって高効率かつ可搬性の高いプログラムを開発できることがわかった．

SXAT と A64FX は他のシステムと比べて行列サイズが小さい場合でも性能が出しやすいことがわかる．最もサ

表 2　使用する実験環境と本論文で用いる名称

Name	System	Processor	SP Peak [TFlops]	DP Peak [TFlops]	Mem. [GB]
A_Intel	ABCI	Intel Xeon Gold 6148@2.40 GHz, 20 core×2	6.14	3.07	384
A_NVIDIA	ABCI	NVIDIA Tesla V100 SXM2	15.6	7.80	16
F_A64FX	FOCUS	Fujitsu A64FX@1.8 GHz, 48 core	5.53	2.76	32
S_Intel	SQUID	Intel Xeon Platinum 8368@2.40 GHz. 38 core×2	11.7	5.84	256
S_AMD	SQUID	AMD EPYC 7402P@2.8 GHz, 24 core	2.15	1.08	128
S_SXAT	SQUID	NEC Type 20A@1.6 GHz, 10 core	6.14	3.07	48

表 3　各環境で使用するソフトウェア

	Compiler	BLAS	VML	Sparse BLAS
A_Intel (MKL)	clang 11.0.1	MKL	MKL	MKL
A_Intel (OpenBLAS)	clang 11.0.1	OpenBLAS	monolish	monolish
A_NVIDIA	clang 11.0.1 (CUDA 10.1)	cuBLAS	monolish	cuSPARSE
F_A64FX	Fujitsu compiler 4.2.1 (based on LLVM 7.1.0)	Fujitsu SSL II	monolish	monolish
S_Intel (MKL)	clang 11.0.1	MKL	MKL	MKL
S_Intel (OpenBLAS)	clang 11.0.1	OpenBLAS	monolish	monolish
S_AMD (OpenBLAS)	clang 11.0.1	OpenBLAS	monolish	monolish
S_SXAT	NEC Compiler 3.2.1	NEC NLC	monolish	monolish

イズが小さい $N = 1,000$ の SGEMM において，他のシステムでは 0.51%から 14.7%しかピーク性能比が出ていないのに対し，S_SXAT はピーク性能の 77.7%, F_A64FX は 42.1%が出ており，絶対性能で比較しても他のシステムより性能が高い．

2 番目にピーク性能比が高いのは A_NVIDIA における $N = 5,000$ の DGEMM で，ピーク性能の 92.4%である．これも十分に高速で OpenMP Offloading によるアクセラレータ向けのプログラムの共通化が性能に与える影響は小さいことが示された．

OpenBLAS を用いた場合の性能はいずれも低く，最大で 44.2%しか出ていない．A_Intel において OpenBLAS をビルドする際のオプションをいくつか変更して試行したが，性能改善効果はほとんど得られなかった．A_Intel および S_Intel は MKL を使用すれば性能が改善したことから，Intel CPU の場合は MKL を使うことで高い性能を出すことができると考えられるが，AMD CPU の場合は課題が残る．OpenBLAS に対する更なる調査が必要であると考えられるが，性能が向上しない場合は AMD CPU に対しては AMD 社の提供している BLAS, LAPACK ライブラリ [21] の使用を検討する必要があると考えられる．

4.3　LU 分解の性能

本節では，$N \times N$ の正方乱数密行列に対する LU 分解の性能を評価する．単精度では，CPU は LAPACK に含まれる sgetrf と sgetrs, GPU では cuSOLVER に含まれる cusolverDnSgetrf, cusolverDnSgetrs を用いる．倍精度では，CPU は LAPACK に含まれる dgetrf と dgetrs, GPU では cuSOLVER に含まれる cusolverDnDgetrf, cusolverDnDgetrs を用いる．行列 A と右辺ベクトル $\boldsymbol{b}$ を関数に渡し，解をベクトル $\boldsymbol{b}$ に上書きする．

各アーキテクチャにおける LU 分解の性能を**表 6**, **表 7** に示す．性能は実行時間から計算量を $2/3N^3$ として求めた．なお，GPU では行列の作成は CPU で行った後に GPU にデータを転送し，結果を CPU に回収するフローを想定している．そのため A_NVIDIA の結果には，行列 A とベクトル $\boldsymbol{b}$ を CPU から GPU に送信し，求解後に解 $\boldsymbol{b}$ を GPU から CPU に送信する時間を含めた．

全体的な傾向として，4.2 節の行列行列積と比べて LU 分解の性能は低いが，適切なベンダ製数値計算ライブラリを使用すれば，F_A64FX 以外の環境ではピーク性能と比べて最大で約 45%の効率が出ており，monolish によって呼び出された LAPACK が十分な性能を発揮できていることがわかる．

4.2 節の結果と同様に，最もピーク性能比が高いのは S_SXAT で，最大で 66.7%のピーク性能比を発揮した．行列行列積の結果では A_NVIDIA は十分に問題サイズが大きければピーク性能の 90%以上を発揮できていたが，LU 分解では最大で約 45.2%にとどまった．

問題サイズが小さい $N = 10,000$ において，単精度 LU 分解の性能が最も高いのは S_Intel の 2.51 TFlops, 倍精度 LU 分解の性能が最も高いのは A_NVIDIA の 1.62 TFlops であった．このように問題サイズや精度によって最も性能の高い環境は異なっており，monolish を用いて可搬性のプログラムを作成し，ユーザが解くべき問題に応じてシステムを切り替えることの意義が示された．

また，4.2 節の結果と比べ，F_A64FX の結果が非常に低く，ピーク性能の 5.15%から 14.7%しか出ていない．環境変数やコンパイルオプションは行列行列積と LU 分解のプログラムで同一であり，大きく性能が低下した原因はわからなかった．これについては，今後も引き続き調査を行う必要があると考えている．

4.4 共役勾配法の性能

本節では，共役勾配法の性能について評価する．2 次元 Laplace 作用素を 5 点中心差分により離散化して得られる次数 $N \times N$ の行列 A を係数とする線型方程式 $A\boldsymbol{x} = \boldsymbol{b}$ を対象とした．行列 A と初期解ベクトル $\boldsymbol{x}$, 右辺ベクトル $\boldsymbol{b}$ を引数にとり，解をベクトル $\boldsymbol{x}$ に上書きする．

なお，LU 分解と同様に，GPU では行列の作成は CPU で行った後に GPU にデータを転送し，結果を CPU に回収するフローを想定している．そのため A_NVIDIA では，行列 A とベクトル $\boldsymbol{x}$, $\boldsymbol{b}$ を CPU から GPU に送信し，求解後に解 $\boldsymbol{x}$ を GPU から CPU に送信する時間を含めた．また，CG 法の関数内で必要なベクトルは monolish によって適切なメモリ空間に自動的に確保するようにしている．

行列の格納形式は，すべて monolish において “computable” 属性をつけて実装されている CRS 形式を用いた．SXAT は著者らの先行研究 [22] によって Block CRS 形式に対する疎行列ベクトルの性能が出ないことが確認されているため高い性能は期待できないが，可搬性を示すために本論文では他の環境と同様に CRS 形式で実行した．SXAT においても，今後 ELL 形式などを実装すれば性能が発揮できると考えている．

反復回数を 1,000 回に固定し，そこから 1 秒間に共役勾配法を性能として用いた．各アーキテクチャにおける性能を**表 8**, **表 9** に示す．

MKL における S_Intel の性能は，サイズが小さい $N = 1,000$ のとき他の環境と比べて 6.8 倍から 79.4 倍高い．これは S_Intel が 114 MB の L3 キャッシュが搭載されているためであると考えられる．S_Intel では単精度では $N = 2,000$ まで，倍精度では $N = 1,000$ までデータがキャッシュに収まる．

同様に，S_AMD の性能も行列行列積や LU 分解の結果と比べて他の環境との性能差が小さく，monolish に実装された疎行列ベクトル積が十分に性能を発揮できていることがわかった．

S_SXAT は他の環境と比べて 3 倍から 100 倍性能が低い．これは事前の予測通り CRS 形式を使用しているためベクトル化が行えないためであると考えられる．これはベクトル化に有効な ELL 形式などを実装する必要がある．

A_NVIDIA の性能は，行列行列積や LU 分解の結果と比べて他の環境との性能差が小さく，特に小さい問題に対しては性能が出ていない．これについて詳細に各関数の時間を計測してみると，サイズの小さい $N = 1,000$ では内積計算の時間が全体時間の 50%を占めた．このことから，コア数の多い GPU では疎行列ベクトル積と比べて内積の集約計算がボトルネックとなり，小さい問題では他の環境より性能が出なにくいことがわかった．また，GPU では ELL 形式や JAD 形式などの格納形式のほうが性能が出やすいことが知られており [23]，SXAT 同様に ELL 形式などを実装することでさらなる高速化が可能と考えている．

A_Intel における MKL と OpenBLAS の性能差は 2%から 20%程度で，性能差は小さい．これは CRS 形式の疎行列ベクトル積が一般的にメモリ性能に成約を受けやすいことから，MKL に実装された疎行列ベクトル積と monolish に実装された疎行列ベクトル積の性能差が小さいためと考えられる．このことから，monolish に実装した疎行列ベクトル積は性能を発揮できていることを確認できた．

5. まとめ

本論文では，プロセッサアーキテクチャの多様化に対し，各プロセッサアーキテクチャで動作する数値計算ライブラリの違いがあることを問題視し，各ベンダやライブラリのデータ型，行列格納形式，データ通信 API などを扱うための統一された API を定義し，1 つのプログラムで様々なアーキテクチャで動作する可搬性の高いオープンソースの線形代数ライブラリを開発することを提案した．

ライブラリを monolish (MONOlithic LIner equation Solvers for Highly-parallel architecture) と命名し，monolish によって統一したい課題として，以下の 4 つを定めた．

課題 1　データ型に依存した関数のインタフェース

課題 2　ホスト，デバイス間のデータ転送 API やメモリ管理

課題 3　数値計算ライブラリのインタフェース

課題 4　行列の構造や格納形式ごとの関数のインタフェース

2.1 節では課題 1 を達成するため，C++のテンプレートや関数オーバロードを用い，型に依存しない BLAS や LAPACK のインタフェースを提案した．

2.2 節および 3 章では課題 2 を達成するため，ホストとデバイスの通信の記述方法やホストメモリとデバイスメモリを管理するためのポインタ型変数の管理について述べた．

CPU 向けにプログラムを GPU で動かすのではなく，GPU 向けに記述したプログラムを CPU でも動くようにする方法をとることで，共有ライブラリを切り替えるだけで簡単にアーキテクチャを切り替えられることや，ベクトルや行列をクラスとしてもち，内部的にホストメモリとデバイスメモリを管理することを提案した．

2.3 節では課題 3 を達成するため，最も機能の多い MKL

表 4 SGEMM の性能 [TFlops] (ピーク比)

	A_Intel (MKL)	A_Intel (OpenBLAS)	A_NVIDIA	F_A64FX	S_Intel (MKL)	S_Intel (OpenBLAS)	S_AMD (OpenBLAS)	S_SXAT
N=1,000	0.91 (14.7)	0.64 (10.4)	0.08 (0.51)	2.33 (42.1)	0.26 (2.23)	0.74 (6.42)	0.25 (11.5)	4.77 (77.7)
N=2,000	2.14 (34.8)	1.33 (21.7)	7.69 (49.3)	3.27 (59.1)	6.31 (54.0)	0.99 (8.36)	0.49 (22.8)	5.46 (88.9)
N=3,000	2.20 (35.8)	1.73 (28.2)	11.2 (71.6)	3.72 (67.3)	8.35 (71.5)	1.10 (9.47)	0.56 (26.1)	5.72 (93.2)
N=4,000	4.12 (66.7)	1.87 (30.5)	12.9 (82.7)	3.37 (61.0)	9.11 (78.0)	1.17 (10.0)	0.63 (29.2)	5.80 (94.4)
N=5,000	4.44 (72.2)	2.15 (35.1)	14.0 (89.7)	3.73 (67.5)	9.55 (81.8)	1.26 (10.8)	0.62 (28.8)	5.84 (95.1)

表 5 DGEMM の性能 [TFlops] (ピーク比)

	A_Intel (MKL)	A_Intel (OpenBLAS)	A_NVIDIA	F_A64FX	S_Intel (MKL)	S_Intel (OpenBLAS)	S_AMD (OpenBLAS)	S_SXAT
N=1,000	0.70 (22.7)	0.59 (19.0)	2.01 (25.8)	1.20 (43.5)	1.41 (24.1)	0.54 (9.23)	0.19 (18.1)	2.76 (89.9)
N=2,000	1.54 (50.0)	0.88 (28.6)	4.49 (57.5)	1.59 (57.3)	3.52 (60.3)	0.62 (10.6)	0.27 (25.4)	2.91 (94.7)
N=3,000	2.01 (65.4)	1.17 (38.1)	6.77 (86.8)	1.74 (62.9)	4.30 (73.6)	0.63 (10.7)	0.26 (24.6)	2.93 (95.3)
N=4,000	2.10 (68.2)	1.28 (41.8)	6.93 (88.9)	1.47 (53.1)	3.49 (59.8)	0.61 (10.5)	0.30 (28.3)	2.96 (96.5)
N=5,000	2.31 (74.8)	1.36 (44.2)	7.21 (92.4)	1.87 (67.6)	4.32 (74.0)	0.61 (10.5)	0.32 (29.8)	2.94 (95.9)

表 6 単精度 LU 分解の性能 [TFlops] (ピーク比), 性能は $2/3N^3$ として求めた.

	A_Intel (MKL)	A_Intel (OpenBLAS)	A_NVIDIA	F_A64FX	S_Intel (MKL)	S_Intel (OpenBLAS)	S_AMD (OpenBLAS)	S_SXAT
N=10,000	1.96 (31.8)	0.67 (11.0)	1.32 (8.53)	0.28 (5.15)	2.51 (21.5)	0.73 (6.34)	0.53 (24.6)	2.14 (34.9)
N=20,000	2.72 (44.2)	1.08 (17.7)	3.93 (25.2)	0.60 (10.8)	4.23 (36.3)	0.96 (8.28)	0.79 (36.6)	3.20 (52.1)
N=30,000	3.47 (56.4)	1.35 (22.0)	5.83 (37.4)	0.49 (8.87)	5.34 (45.8)	1.06 (9.12)	0.91 (42.3)	3.75 (61.1)

表 7 倍精度 LU 分解の性能 [TFlops] (ピーク比), 性能は $2/3N^3$ として求めた.

	A_Intel (MKL)	A_Intel (OpenBLAS)	A_NVIDIA	F_A64FX	S_Intel (MKL)	S_Intel (OpenBLAS)	S_AMD (OpenBLAS)	S_SXAT
N=10,000	1.27 (41.5)	0.41 (13.2)	1.62 (20.8)	0.35 (10.9)	1.39 (23.8)	0.45 (7.61)	0.38 (35.4)	1.34 (43.6)
N=20,000	1.61 (52.3)	0.62 (19.6)	2.81 (36.0)	0.31 (10.8)	2.19 (37.5)	0.52 (8.93)	0.46 (42.5)	1.81 (58.9)
N=30,000	1.78 (57.9)	0.76 (24.6)	3.53 (45.2)	0.41 (14.7)	2.61 (44.6)	0.57 (9.85)	0.49 (45.9)	2.05 (66.7)

表 8 1 秒間で計算できた単精度前処理なし CG 法の反復回数

	A_Intel (MKL)	A_Intel (OpenBLAS)	A_NVIDIA	F_A64FX	S_Intel (MKL)	S_Intel (OpenBLAS)	S_AMD (OpenBLAS)	S_SXAT
N=1,000	374	330	203	538	3,573	401	354	45
N=2,000	89	71	171	87	715	89	94	11
N=3,000	31	30	146	39	208	33	44	5

表 9 1 秒間で計算できた倍精度前処理なし CG 法の反復回数

	A_Intel (MKL)	A_Intel (OpenBLAS)	A_NVIDIA	F_A64FX	S_Intel (MKL)	S_Intel (OpenBLAS)	S_AMD (OpenBLAS)	S_SXAT
N=1,000	440	428	483	203	2,661	224	248	44
N=2,000	93	87	166	50	274	42	77	11
N=3,000	42	38	138	23	108	17	37	5

を参考に，実装すべき関数を以下の 5 つのグループにわけて検討した．最もユーザが多いと考えられる MKL, CUDA ライブラリの呼び出しを実装し，MKL や CUDA ライブラリが使用できない環境では，BLAS, LAPACK に含まれない関数は monolish に実装された汎用プログラムに切り替えることで，様々なアーキテクチャでも動作するための実装の切り替え方法を提案した．

2.4 節では，疎行列の各格納形式は性能を出しやすい演算が異なることから，格納形式を “editable” と “computable” の 2 つの属性に分けて考えることを提案した．

“editable” は行列要素への参照や編集ができることを意味し，“computable” は並列化された行列計算や，アクセ

ラレータでの計算ができることを意味する．このように属性を分けて機能を定義することで，ユーザに対して高い性能を保証しやすくすることができることを示した．

最後に 4 章で，開発した monolish を 8 つの環境で評価した．行列行列積，LU 分解，CG 法のプログラムを対象に実験を行い，プログラムを変更せず，コンパイラなどを変更するだけですべての環境で動作させることができた．

行列行列積に対する実験では，最もピーク性能比が高いものでピーク性能の 95.9%を発揮することができた．このことから，C++を用いて BLAS の型依存を隠蔽し，内部でベンダ製数値計算ライブラリを呼び分けることのオーバヘッドは小さく，本研究の提案によって高効率かつ可搬性の高いプログラムを開発できることがわかった．

LU 分解や共役勾配法に対する実験では，問題サイズや精度によって最も性能の高い環境が異なった．monolish を用いて可搬性のプログラムを作成し，ユーザが解くべき問題に応じて環境を切り替えることの意義が示された．

今後の課題として，(1) OpenMP Offloading を用いて実装した axpy は CUDA から cuBLAS を呼び出した場合と比べて約 10 %性能が低かった．これは良い結果だと考えられるが，コンパイルオプションや指示句によるさらなる高速化の方法を調査すること，(2) 性能がでなかった AMD CPU に対し AMD 製の BLAS の性能を評価すること, (3) 性能がでなかった A64FX の LU 分解，及び共役勾配法の性能を分析すること，(4) SXAT や GPU のためにベクトル化に有利な ELL 形式などを実装することが挙げられる．特に (3) については，行列行列積の性能は出ているが，LAPACK の LU 分解の性能が出ていない結果には疑問が残る．環境変数やオプションなどを調査し，改善する必要があると考えられる．

monolish は `https://github.com/ricosjp/monolish` より利用可能である．今回取り上げた機能以外にも，固有値ソルバなど様々な機能が利用できる．

参考文献

[1] FUJITSU, "FUJISTU Processor A64FX." https://www.fujitsu.com/jp/products/computing/servers/supercomputer/a64fx/, (参照 2021-06-19).

[2] NEC, "NEC SX-Aurora TSUBASA Documentation." https://www.hpc.nec/documents/, (参照 2021-06-19).

[3] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, "An Extended Set of FORTRAN Basic Linear Algebra Subprograms," *ACM Transactions on Mathematical Software*, vol. 14, no. 1, pp. 1–17, 1988.

[4] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, and A. McKenney, "Lapack users' guide," *SIAM*, 1999.

[5] R. Barrett, M. W. Berry, T. F. Chan, J. W. Demmel, J. Donato, J. J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst, "Templates for the solution of linear systems: building blocks for iterative methods," pp. 55–65, SIAM, 1994.

[6] Exascale Computing Project, "SLATE (Software for Linear Algebra Targeting Exascale)." http://icl.utk.edu/slate/, (参照 2021-06-19).

[7] M. Gates, J. Kurzak, A. Charara, A. YarKhan, and J. Dongarra, "Slate: Design of a modern distributed and accelerated linear algebra library," pp. 1–18, 2019.

[8] J. Sanders and E. Kandrot, "Cuda by example: an introduction to general-purpose gpu programming," 2010.

[9] 大阪大学 サイバーメディアセンター 大規模計算システム, "SQUID (Supercomputer for Quest to Unsolved Interdisciplinary Datascience)." http://www.hpc.cmc.osaka-u.ac.jp/squid/, (参照 2021-06-19).

[10] Intel, "The New Strided API for the Intel Math Kernel Library (MKL) Vector Math (VM) component." https://software.intel.com/content/www/us/en/develop/articles/new-mkl-vml-api.html, (参照 2021-06-19).

[11] Y. Saad, "Iterative Methods for Sparse Linear Systems," pp. 151–243, SIAM, 2003.

[12] Scipy developers, "SciPy, Python-based ecosystem of open-source software for mathematics, science, and engineering." https://www.scipy.org/, (参照 2021-06-19).

[13] Advanced Micro Devices, Inc., "HIP Programming Guide." https://rocmdocs.amd.com/en/latest/Programming_Guides/HIP-GUIDE.html, (参照 2021-06-19).

[14] S. F. Antao, A. Bataev, A. C. Jacob, G.-T. Bercea, A. E. Eichenberger, G. Rokos, M. Martineau, T. Jin, G. Ozen, and Z. Sura, "Offloading support for openmp in clang and llvm," in *2016 Third Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, pp. 1–11, IEEE, 2016.

[15] clang11 documentation, "OpenMP Support." https://releases.llvm.org/11.0.0/tools/clang/docs/OpenMPSupport.html, (参照 2021-06-19).

[16] 国立研究開発法人産業技術総合研究所, "AI 橋渡しクラウド (AI Bridging Cloud Infrastructure、ABCI)." https://abci.ai/ja/, (参照 2021-06-19).

[17] 計算科学振興財団, "FOCUS スパコンシステム." https://www.j-focus.or.jp/focus/, (参照 2021-06-19).

[18] Y. Yamada and S. Momose, "Vector engine processor of nec's brand-new supercomputer sx-aurora tsubasa," in *Proceedings of A Symposium on High Performance Chips, Hot Chips*, vol. 30, pp. 19–21, 2018.

[19] Z. Xianyi, "OpenBLAS." https://www.openblas.net/, (参照 2021-06-19).

[20] 大島 聡史, 永井 亨, 片桐 孝洋, "スーパーコンピュータ「不老」の性能評価," **情報処理学会研究報告**, vol. 2020-HPC-175, no. 17, pp. 1–10, 2020.

[21] Advanced Micro Devices, Inc., "BLAS Library." https://developer.amd.com/amd-aocl/blas-library/, (参照 2021-06-19).

[22] 菱沼 利彰, 井原 遊, 高村 守幸, 平野 哲, 萩原 孝, 岩田 直樹, 奥田 洋司, "Sx-aurora tsubasa における有限要素解析のための共役勾配法の性能評価," **情報処理学会研究報告**, vol. 2020-HPC-175, no. 18, pp. 1–10, 2020.

[23] S. Filippone, V. Cardellini, D. Barbieri, and A. Fanfarillo, "Sparse matrix-vector multiplication on gpgpus," *ACM Transactions on Mathematical Software (TOMS)*, vol. 43, no. 4, pp. 1–49, 2017.

付　録

```
#include "monolish_blas.hpp"
#include "monolish_equation.hpp"
#include "monolish_vml.hpp"

template <typename MAT, typename Float>
void cg(const MAT &A, monolish::vector<Float> &x, const monolish::vector<Float> &b, Float tol)
    {
  monolish::vector<Float> r(A.get_row(), 0.0);
  monolish::vector<Float> p(A.get_row(), 0.0);
  monolish::vector<Float> q(A.get_row(), 0.0);
  r.send(); p.send(); q.send();

  monolish::blas::matvec(A, x, q);
  monolish::vml::sub(b, q, r);
  monolish::blas::copy(r, p);

  for (size_t iter = 0; iter < A.get_row(); iter++) {
    monolish::blas::matvec(A, p, q); // q = Ap
    auto tmp = monolish::blas::dot(r, r);
    auto alpha = tmp / monolish::blas::dot(p, q);
    monolish::blas::axpy(alpha, p, x); // x = alpha*p + x
    monolish::blas::axpy(-alpha, q, r); // r = -alpha * q + r
    auto beta = monolish::blas::dot(r, r) / tmp; // beta = (r, r)
    monolish::blas::xpay(beta, r, p); // p = r + beta*p
    auto resid = monolish::blas::nrm2(r); // check convergence
    if (resid < tol) { return; }
  }
}

int main() {
  // create 2D 5point laplacian matrix stored in COO
  int size = 100;
  monolish::matrix::COO<double> COO = monolish::util::laplacian_matrix_2D_5p<double>(size,
      size);

  monolish::matrix::CRS<double> A(COO); // create CRS matrix from COO
  monolish::vector<double> x(A.get_row(), 0.0, 1.0); // initial x is rand(0~1)
  monolish::vector<double> b(A.get_row(), 1.0); // initial b is {1, 1, 1, ...,1}
  A.send(); x.send(); b.send(); // send A, x, b to GPU device

  cg(A, x, b, 1.0e-12); // solve

  x.recv(); // recv x from GPU device

  return 0;
}
```

図 **A·1** monolish を用いて実装した共役勾配法のプログラム

ⓒ 2021 Information Processing Society of Japan