B-019

単一 GPU 環境における NumPy 互換ライブラリを用いた Python プログラミングの比較

A Comparison of Python Programming with NumPy Compatible Libraries in a Single GPU Environment

田道 竜大 †　藤本 典幸 †
Ryuto Tamichi　Noriyuki Fujimoto

1 はじめに

Python は，データサイエンス，機械学習や数値計算などで広く用いられる言語である．Python はインタプリタ型の言語であり，C/C++や Fortran などコンパイル型言語と比較すると実行速度の面で大幅に劣る．しかし，C/C++などで記述され高速化された様々なライブラリが利用できるため，比較的簡潔な記述が可能で，かつ実用的な実行速度を実現している．特に，NumPy[1] や SciPy[2] といった，多次元配列や数値計算を扱うライブラリは広く用いられている．データサイエンスや機械学習の分野では，膨大な数のデータセットが格納された行列やテンソルを扱う演算を多用するため，近年は GPU（Graphics Processing Unit）が持つ多数の演算資源を用いた並列計算による，プログラム実行時間の高速化が必要とされている．NVIDIA 社の CUDA[3] は，GPU 向けプログラミングモデルの一つであり C/C++ベースで記述できる．しかしながら，CUDA は GPU の各コアに割り当てられるスレッドが担当する処理を，カーネルと呼ばれる関数に記述する必要があり，プログラミング手法の習得には時間を要する．

そこで，比較的容易に Python プログラムを GPU 上で実行し高速化する方法の一つとして，CUDA 実装された NumPy 互換ライブラリの利用が挙げられる．この方法では，既存の NumPy を import 文で互換ライブラリに置き換えるのみで GPU を利用できる．代表的な NumPy 互換ライブラリとして Preferred Networks 社の CuPy[4] と，2021 年に NVIDIA 社が発表した cuNumeric[5] がある．cuNumeric はまだ β 版のみのリリースだが CuPy と比較して，複数の GPU が利用できるマルチ GPU 環境に対応しており，よりスケーラブルに高速化性能を発揮する．しかしながら，GPU1 基のみの単一 GPU 環境における性能については，CuPy のほうが高い性能を発揮するとされている [6]．この点については，NVIDIA の公式から詳細なテストデータは公表されていない．そのため，本論文では複数のベンチマークプログラムを用いて，単一 GPU 環境における cuNumeric と CuPy の性能を比較する．

2 準備

2.1 GPU の構造と性質

Graphics Processing Unit（GPU）は，元々グラフィックスレンダリングをより高速に計算するための専用アクセラレータとして開発された．一方，時間の経過と共にグラフィックスレンダリングに求められる機能が複雑化し，GPU の演算機能の柔軟性が増した結果，現在の GPU は並列計算能力の高いプロセッサとして構成されている [3]．CPU は多数の演算命令を持つ多機能で高速なコアを少量実装しているのに対して，GPU は限られた機能を持つ小型なコアを多数実装している．GPU のコア全体で十分な量の演算命令を実行できれば CPU よりも優れた演算性能を得られる一方，一般的に GPU のコア単体の性能は CPU よりも低くなる．従って GPU の演算性能を引き出すには，データ依存関係がなく並列実行可能な計算を大量に用意して，GPU のコアに供給する必要がある．また，メモリの帯域幅が CPU-主記憶バスは数十 GB/sec オーダー，CPU-VRAM 間の PCI Express バスは数十 GB/sec オーダーと比較して，GPU-ビデオメモリ（VRAM）バスでは数百・数千 GB/sec オーダーと大きな差があり，主記憶-VRAM 間のメモリ転送はボトルネックとなる．多数の GPU コアを用いた並列処理の効果を最大限発揮するには，このメモリ転送のボトルネックに起因するオーバーヘッドを最小限に抑えなければならない．

2.2 cuNumeric

cuNumeric は，NVIDIA 社が開発している CUDA 実装の NumPy 互換ライブラリである．執筆時点では β 版がリリースされており，Ver.23.03 が最新リリースである．

2.2.1 cuNumeric による GPU 実装手法

cuNumeric は，図 1 に示すように既存の NumPy で記述されたコードの import 文を書き換え，NumPy の代わりに cuNumeric をインポートすることで，GPU 実装された NumPy モジュールに置き換えることができる．NumPy で提供される多次元配列データ構造，ndarray（`numpy.ndrray`）やそれを利用する関数群が cuNumeric への置き換えに対応している．図 1 の例では，A は乱数を各要素とする M × N の ndarray とし，A を 0.5 倍した ndarray である B との行列積 C を求めている．

2.2.2 cuNumeric の構成

cuNumeric による GPU 実装版 NumPy モジュールの構成について説明する．まず，既存の NumPy は C/C++や Fortran で実装され，ビルドされたものを呼び出す.py ファイルがモジュールとして構成されている．この仕組みより，cuNumeric では NumPy モジュールの各機能を CUDA で GPU 実装し，ビルドされたものを呼び出す.py ファイルを cuNumeric モジュールとして構成する．各 NumPy モジュールと同名の cuNumeric モ

†大阪公立大学 大学院情報学研究科 Graduate School of Informatics, Osaka Metropolitan University

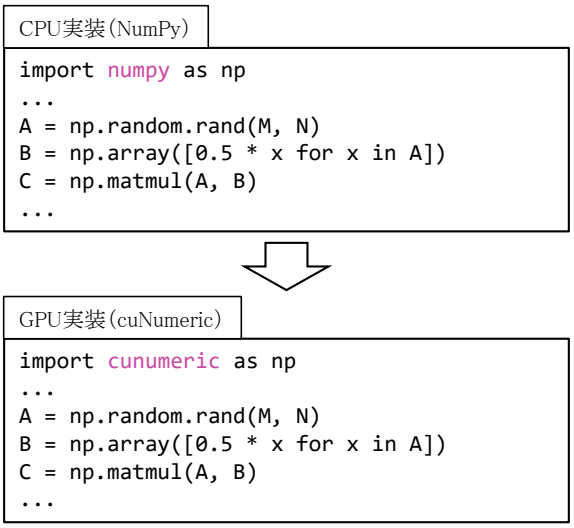


図 1: NumPy から cuNumeric への書き換え

ジュールを呼び出すことにより，既存の NumPy 実装されたコードを書き換えることなく，cuNumeric から GPU 実装された各モジュールを利用できる．cuNumeric はまだ β 版であることから，GPU 実装に未対応の NumPy モジュールも多数存在する．GPU 実装に未対応のモジュールも cuNumeric から呼び出すことはできるが，その場合には既存の NumPy モジュールを自動的に呼び出して実行する．cuNumeric ではこの操作を Fallback と呼んでいる．個々のモジュールや関数の cuNumeric 対応状況については，公式の API リファレンス [7] を参照されたい．

2.3 Legate

Legate[8] は，cuNumeric を用いた Python プログラムの実行に必要な環境である．

2.3.1 Legate の概要

Legate は，複数個の GPU が利用できるマルチ GPU 環境や，複数の GPU ノードから構成されるスーパーコンピュータといった高性能計算環境で Python プログラムの並列分散処理を可能にするエコシステムである [9]．NVIDIA 社が開発中であり，cuNumeric は Legate の一部として提供されている．そのため，cuNumeric を利用する Python プログラムの実行には必須の環境である．Legate 環境では，`legate` コマンドにより Python プログラムを実行できる．`legate` コマンドは，実行環境を準備した後，Python インタプリタを呼び出してプログラムを実行するスクリプトとして構成されており，公式ではこれを driver script と呼んでいる [8]．`legate` コマンドのオプションにより，実行に利用する CPU/GPU の数や主記憶/VRAM の割り当て量を設定できる．例えば，cuNumeric を用いた Python プログラム `test.py` を Legate 環境で実行したい場合は，

```
$ legate --gpus 1 --fbmem 23552 test.py
```

とコマンドをシェル上で実行すればよい．`--gpus` オプションでは Legate 環境で割り当てる GPU の数を指定し，`--fbmem` オプションでは Legate 環境に割り当てる VRAM 量（単位：MB）を指定している．この例では割り当て GPU 数 1，割り当て VRAM 数 23,552MB=23GB としている．

2.1 節で述べたように，既存の CPU 向けに記述されたプログラムを GPU 実装して十分な高速化性能を得るためには，GPU の各コアに十分な量の演算処理を与える必要がある．そのため，既存の NumPy で実装された Python プログラムを cuNumeric を用いて GPU 実装する際にもこの問題が生じる．特に，サイズの小さい NumPy 配列（ndarray）を扱う場合は cuNumeric を用いるよりも，既存の NumPy で実装するほうが CPU-GPU 間のメモリ転送にかかるオーバーヘッドがない為，高速となる．

2.3.2 cuNumeric 向け実行時パラメータ

Legate では，cuNumeric を用いた Python プログラムを実行する際に，`legate` コマンドのオプションとは別に環境変数として与えられる各種パラメータがあり，実行するプログラムに応じて調整することにより最適な性能が得られる．但し，NVIDIA 社が公式にこれらのパラメータについてはアナウンスしておらず，実験段階の機能とされる．その一覧とデフォルト値を表 1 に示す．表 1 で挙げたパラメータは，cuNumeric Ver.23.03 で提供されているものの中で実行性能に関係するものである．

`CUNUMERIC_MIN_GPU_CHUNK` は，cuNumeric を用いるプログラムで GPU を利用して処理する最小の配列サイズである．プログラム中で処理対象となる ndarray の長さがこの値以上である場合のみ，GPU 実装された cuNumeric の関数で実行する．サイズが設定値より小さい ndarray の場合は，通常の NumPy 実装へ Fall-Back が行われる．すなわち，このパラメータにより GPU 実装の弱点であるサイズが小さい問題に対しては cuNumeric の GPU 実装が用いられず，CPU 上で処理することでプログラムの実行速度の低下を防ぐことができる．`CUNUMERIC_MIN_OMP_CHUNK` は，ndarray に関する処理が CPU で実行される場合において，ndarray の長さがこの値以上の場合は OpenMP による並列処理が有効になる．GPU で実行する場合と同様にマルチコア CPU で OpenMP を用いて NumPy の処理を並列化する際にも，サイズが小さい問題に対しては並列化にかかる処理がオーバーヘッドとなり，実行速度の低下を招く．このパラメータをプログラムに応じて調整することで，NumPy 関数の OpenMP 並列化で実行速度が低下することを防ぐことができる．`CUNUMERIC_PRELOAD_CUDALIBS` は，cuNumeric を用いたプログラムを実行する際，プログラム本体の実行前に CUDA 関係のライブラリをプリ

表 1: 主な Legate 実行時パラメータ

パラメータ名	デフォルト値
CUNUMERIC_MIN_GPU_CHUNK	65536
CUNUMERIC_MIN_OMP_CHUNK	8192
CUNUMERIC_PRELOAD_CUDALIBS	False
CUNUMERIC_FAST_MATH	False

ロードし，あらかじめ初期化するか否かを True, False で選択するパラメータである．CUNUMERIC_FAST_MATH は Ampere アーキテクチャ以降の TensorCore[10] を搭載した GPU でこのパラメータを True とすると，cuNumeric の実装で用いられている線形代数ライブラリ cuBLAS[11] において，TF32 フォーマット [12] による単精度浮動小数点演算の高速化をサポートする．例えば，cuNumeric を用いたプログラム test.py において ndarray のサイズに依らず常に GPU 上で処理し，TF32 フォーマットによる高速化を有効にしたい場合は，

```
$ CUNUMERIC_MIN_GPU_CHUNK=1 \
  CUNUMERIC_FAST_MATH=True \
  legate --gpus 1 --fbmem 23552 test.py
```

とコマンドを実行すればよい．この際の注意が，Legate の実行時パラメータは legate コマンドの実行プロンプトのみで有効な環境変数であり，legate コマンドの直前部分で毎回指定する必要がある．

2.3.3 docker 環境での実行

Legate は docker[13] 環境で実行が可能であり，本論文では独自に実験用の docker イメージ nfujimoto/cunumeric:conda-v4 を構築し，以後このイメージを用いてベンチマークを行うものとする．この実験用 docker イメージは，Ubuntu 22.04 ベースのイメージ上に cuNumeric を含む Legate 環境が構築されている．cuNumeric を用いたプログラム test.py を docker 環境上で実行する場合は，

```
$ docker run --gpus all --rm \
  nfujimoto/cunumeric:conda-v4 \
  legate --gpus 1 --fbmem 23552 test.py
```

とコマンドを実行する．特に，docker はマルチ GPU 環境に対応しており--gpus all オプションで，ホスト PC 上に存在するすべての GPU を割り当てることができる．また，インストールされている Python インタプリタは Python3.11.2 である．

2.4 CuPy

CuPy は，Preferred Networks 社が開発している CUDA 実装の NumPy 互換ライブラリで，cuNumeric と同様に import 文で NumPy の代わりにインポートすることで，GPU に対応した NumPy のモジュールが利用できる．図 2 は図 1 と同じく，import 文により NumPy から CuPy へ置き換えを行う様子を示している．cuNumeric は 2021 年に発表されたが CuPy はそれ以前から開発されており，cuNumeric よりも多数の NumPy モジュールが GPU 実装されている．また，cuNumeric のように Legate など特別な環境を必要とせず，CUDA に対応した GPU 環境があれば通常の Python コマンドで実行できる．ただし，cuNumeric のようにマルチ GPU 環境への対応や，GPU 実装に未対応なモジュールの FallBack 機能はサポートされていない．CuPy は Legate 環境でも実行がサポートされているが，cuNumeric のように実行時パラメータは提供されておらず，プログラムの問題サイズに合わせた GPU 実装と CPU 実装の性能最適化には未対応である．すなわち，プログラムにおける ndarray のサイズに依らず常に GPU 実装の NumPy モジュールが呼び出される．そのため，ndarray のサイズが小さいプログラムでは 2.1 節で述べたように，主記憶-VRAM 間のメモリ転送時間がボトルネックとなり GPU 実装による並列化の効果が小さくなるか，かえってオーバーヘッドとなりうる．

CPU実装（NumPy）

```
import numpy as np
...
A = np.random.rand(M, N)
B = np.array([0.5 * x for x in A])
C = np.matmul(A, B)
...
```

GPU実装（CuPy）

```
import cupy as np
...
A = np.random.rand(M, N)
B = np.array([0.5 * x for x in A])
C = np.matmul(A, B)
...
```

図 2: NumPy から CuPy への書き換え

3 評価実験

3.1 ベンチマーク問題

cuNumeric, CuPy それぞれの性能を比較するために，Legate の公式 GitHub 上で公開されているベンチマーク問題 [14] のうち，実際に NumPy, cuNumeric, CuPy のそれぞれすべての実装で実行でき，その実行時間を計測可能な 12 のプログラムを使用した．その一覧と以後使用する問題番号の対応表を表 2 に示す．ここで示す問題番号は便宜上定義する本論文固有の番号である．

以下，それぞれのプログラムの特徴について説明する．

表 2: ベンチマークプログラム一覧

問題番号	プログラム名
P1	black_scholes.py
P2	gemm.py
P3	jacobi.py
P4	kmeans.py
P5	kmeans_slow.py
P6	linreg.py
P7	logreg.py
P8	lstm_backward.py
P9	lstm_forward.py
P10	scan.py
P11	sort.py
P12	stencil.py

P1 は Black–Scholes 方程式の数値解を求めるプログラムである．P2 は，行列 A, B, C とスカラー α, β に対して $C = \alpha AB + \beta C$ を計算する Gemm 演算のプログラムである．これら 2 つの問題は if 文などの複雑な制御構造を持たず，すべての処理が NumPy の関数と ndarray の演算で構成されている．P3 は Jacobi 反復法で連立一次方程式を解くプログラムで，ndarray の演算を既定の回数だけ for 文で反復処理する．既定の反復回数は 1,000 回である．P4, P5 は k-means 法でクラスタリングを行うプログラムで，P4 と P5 の違いはクラスター重心を計算するアルゴリズムの違いである．P5 のほうが重心計算に用いる NumPy 関数の呼び出し回数が多い実装となっている．P6，P7 はそれぞれ線型回帰，ロジスティック回帰のプログラムである．P6 と P7 ともに既定の反復回数だけ回帰式の計算を行う実装となっており，元々の回帰式に用いる関数の違いから，ロジスティック回帰の P7 のほうが呼び出される NumPy 関数や ndarray の演算数は多い．既定の反復回数は 1,000 回である．P8, P9 は再帰型ニューラルネットワークアーキテクチャの一つである Long Short-Term Memory（LSTM）を実装するプログラムである．P8 は逆伝播，P9 は順伝播を計算し，指定したサイズの ndarray で与えられる正規乱数列に対して LSTM のシミュレートを行う．if 文などの制御構造は比較的少ないが，ndarray の各要素に対して一括で操作するスライス演算を多用している．P10 は，接頭辞和（Prefix-Sums）を計算するスキャン操作を実装したプログラムである．指定したサイズの ndarray に対して，数種類の NumPy 関数と Python の組み込み関数 `getattr` で実装されている．P11 は，NumPy の `sort` 関数を用いて指定したサイズの ndarray をソートするプログラムである．NumPy の関数のみで実装されている．P12 は，指定されたサイズの格子点領域で指定された回数だけ反復して，2 次元 5 点ステンシル計算を行うプログラムである．複雑な制御構造はなく 2 次元 ndarray の各要素を近傍の平均値と入れ替える操作のみである．

3.2 実験環境

実験に用いた計算機の仕様は表 3 に示す通りである．この実験では，単一 GPU 環境における cuNumeric と CuPy による GPU 実装の性能を比較するため，2.3.3 節で説明した実験用 docker イメージを用いる．また，cuNumeric 版は `legate` コマンド，NumPy 版と CuPy 版は `python` コマンドで実行する．`legate` コマンドでは，表 4 に示すパラメータとオプションを実行時に指定する．デフォルト値の場合は default としている．2.4 で説明したように，CuPy は問題サイズに依らず常に GPU 上で処理を行うため，cuNumeric でも同様の条件で比較するべく，`CUNUMERIC_MIN_GPU_CHUNK=1` としている．また，Legate の実行時パラメータが実験環境でどの範囲で設定可能かを事前に検証したところ，本実験環境では `CUNUMERIC_PRELOAD_CUDALIBS=True` に設定した場合，プログラムの実行前に VRAM のオーバーフローを示すエラーとなったため CUDA ライブラリのプリロードは行わないこととした．`--fbmem` オプションの値については，本実験で用いる GPU の VRAM は 24 GB であることから，Legate に割り当てられるサイズを検証したところ約 22.8 GB まではエラーとならず割り当てられることが判明した．今回は安全マージンをとり，22.5 GB（=23,040 MB）に設定した．

表 3: 実験用計算機の仕様

項目	仕様
CPU	AMD Ryzen 9 7950X （16 物理コア，32 論理コア）
主記憶	64 GB
GPU	NVIDIA GeForce RTX 4090 （16,384 CUDA コア）
VRAM	24 GB
OS	Rocky Linux release 8.7

表 4: 実験用 Legate パラメータとオプション

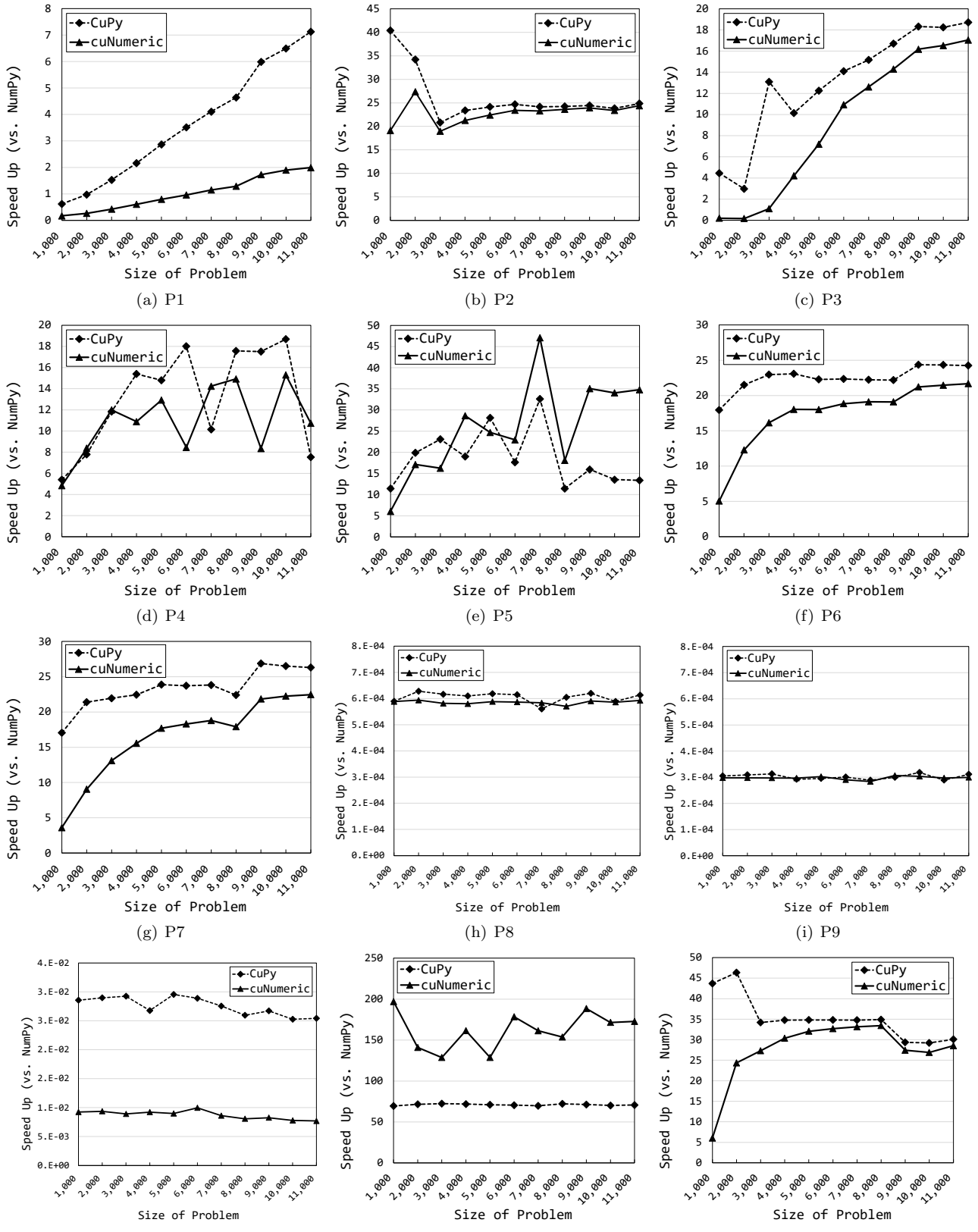
パラメータ名	設定値
`CUNUMERIC_MIN_GPU_CHUNK`	1
`CUNUMERIC_MIN_OMP_CHUNK`	default
`CUNUMERIC_PRELOAD_CUDALIBS`	False
`CUNUMERIC_FAST_MATH`	True
`--gpus`	1
`--fbmem`	23040

3.3 計算時間の比較

3.1 節で説明した各ベンチマークプログラムの実行時間を，NumPy, cuNumeric, CuPy それぞれの実装で測定する実験を行った．実験は，各ベンチマークプログラムで使用されている ndarray のサイズを変化させることにより，問題サイズが実行時間に与える影響について測定・比較した．ベンチマーク問題における NumPy 版, cuNumeric 版, CuPy 版の各実行時間を，問題サイズ 1,000〜11,000 で測定し，NumPy 版の結果を 1 としたときの速度向上率のグラフを図 3 に示す．このグラフにプロットされた値が 1 以下の場合は cuNumeric, CuPy を用いた GPU 実装により実行速度は向上しなかったことを表している．P8, P9, P10 においては，全問題サイズにて速度向上率が 1 以下であり，GPU 実装がかえって性能の低下を招いたといえる．P5 においては概ね，P11 では全問題サイズにおいて cuNumeric のほうが CuPy よりも速度向上率が高くなった．特に，P11 は `sort` 関数単体の性能を計測しており，cuNumeric 版の `cunumeric.sort` 関数は最大で約 200 倍近い高速化性能を有しているといえる．それ以外の問題では，CuPy のほうが cuNumeric よりも速度向上率が高い結果となった．

3.4 メモリ転送命令の割合の比較

3.3 節で述べた実験結果より，GPU 実装の NumPy 互換ライブラリを用いることでかえって性能が低下する場合があった．また，cuNumeric はほとんどの場合において CuPy よりも高速化性能が低いという結果になった．特に，GPU 上で処理するプログラムでは主記憶-VRAM



(a) P1 (b) P2 (c) P3

(d) P4 (e) P5 (f) P6

(g) P7 (h) P8 (i) P9

(j) P10 (k) P11 (l) P12

図 3: 各ベンチマーク問題における速度向上率（NumPy 比）

間のデータ転送がボトルネックとなり，実行速度の低下を招くことが多い．そこで，cuNumeric と CuPy から呼び出される GPU のカーネル関数において，メモリ転送命令の割合がどの程度占めるのかを調査した．実験用 docker イメージ上で構成されている CUDA プログラムのプロファイラ，Nsight System[15] を用いて，各ベンチマーク問題実行時に呼び出されるメモリ転送命令の割合をプロファイリングした．その結果は表 5 に示す通りである．表 5 の結果と 3.3 節で述べた結果より，CuPy のほうが cuNumeric よりも速度向上率が高い問題については概ね，CuPy のほうがメモリ転送命令の呼び出し割合が低いといえる．また，cuNumeric と CuPy でともに性能が低下した P8, P9 においては，cuNumeric, CuPy どちらもメモリ転送命令の割合が比較的高いことがわかる．P8, P9 は LSTM のアルゴリズムの特性から，元々の計算量が低いため GPU による並列化の性能が発揮できなかったと考えられる．さらに，全体を通して cuNumeric のメモリ転送命令の割合が高いことは，Legate がマルチ GPU 環境においてスケーラブルな性能を発揮することを目的としており，ノード間（GPU 間）でデータを共有・分配するために主記憶-VRAM 間のやり取りが多くなる実装となっているのではないかと考える．そのため，今回の単一 GPU 環境では複数の NumPy 関数を呼び出す場合にはメモリ転送のオーバーヘッドが顕著に現れたと考えられる．この問題の対処法としては，2.3.2 節で説明した `CUNUMERIC_MIN_GPU_CHUNK` の値を，問題に応じて設定することが挙げられる．これにより，cuNumeric 版で実行速度が低下するプログラムについて，問題サイズが GPU のコア数と比べて小さい場合は CPU で処理することで，プログラムとしての性能を向上させることができる．この点は，CuPy 単体では実現できないため cuNumeric を含む Legate の大きな利点であるといえる．

4 おわりに

本論文では，CUDA 実装の NumPy 互換ライブラリである cuNumeric と CuPy の性能について，複数のベンチマーク問題により比較した．実行時間では概ね CuPy 版のほうが高速であるという結果となったが，`sort` 関数については cuNumeric が NumPy 版と比較して最大で約 200 倍の性能を発揮した．また，メモリ転送命令の呼び出し割合を調査したところ NumPy 版よりも実行速度が劣る場合には，比較的多くのメモリ転送を行っていることがわかった．特に，cuNumeric では全体的にメモリ転送命令の呼び出し割合が高く，単一 GPU 環境において CuPy よりも実行速度が劣る原因の一つだと考えられる．これらの問題を解消し，cuNumeric 実装における性能を向上させる方法の一つとして，Legate の実行時パラメータを問題に合わせて最適化することが挙げられる．今後の課題として，Legate の実行時パラメータを最適化する方法を検討する必要がある．

表 5: 各問題におけるメモリ転送命令の割合

問題番号	メモリ転送命令の割合［%］	
	cuNumeric	CuPy
P1	4.9	0
P2	12.7	0.2
P3	1.7	0.1
P4	1.5	1.7
P5	5.6	1.6
P6	10.1	0.1
P7	6.8	0.1
P8	7	3.1
P9	3.9	10.5
P10	0.4	1.7
P11	1.6	3.6
P12	9.5	0.2

謝辞

本研究は JSPS 科研費 20K11842 と大阪公立大学戦略的研究推進事業の助成を受けたものです．

参考文献

[1] NumPy Project: “NumPy”, `https://numpy.org/`.

[2] SciPy Project: “SciPy”, `https://scipy.org/`.

[3] NVIDIA Corporation: “CUDA Toolkit”, `https://developer.nvidia.com/cuda-toolkit`.

[4] Preferred Networks, Inc.: “CuPy”, `https://cupy.dev/`.

[5] NVIDIA Corporation: “cuNumeric”, `https://developer.nvidia.com/cunumeric`.

[6] M. Bauer and M. Garland: “Legate NumPy: Accelerated and Distributed Array Computing”, Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC ’19, New York, NY, USA, Association for Computing Machinery (2019).

[7] NVIDIA Corporation: “cuNumeric API Reference”, `https://nv-legate.github.io/cunumeric/23.03/api/`.

[8] NVIDIA Corporation: “Legate”, `https://nv-legate.github.io/legate.core/README.html`.

[9] M. Bauer, W. Lee, M. Papadakis, M. Zalewski and M. Garland: “Supercomputing in Python With Legate”, Computing in Science and Engineering, **23**, 4, pp. 73–79 (2021).

[10] NVIDIA Corporation: “NVIDIA H100 Tensor Core GPU Architecture”, `https://resources.nvidia.com/en-us-tensor-core`.

[11] NVIDIA Corporation: “cuBLAS”, `https://developer.nvidia.com/cublas`.

[12] J. Choquette, W. Gandhi, O. Giroux, N. Stam and R. Krashinsky: “NVIDIA A100 Tensor Core GPU: Performance and Innovation”, IEEE Micro, **41**, 2, pp. 29–35 (2021).

[13] Docker Inc.: “Docker”, `https://www.docker.com/`.

[14] NVIDIA Corporation: “cunumeric/examples”, `https://github.com/nv-legate/cunumeric/tree/branch-23.03/examples`.

[15] NVIDIA Corporation: “NVIDIA Nsight Systems”, `https://developer.nvidia.com/nsight-systems`.