医用画像処理における LDDMM のマルチ GPU 高速化

杉浦拓未 [1]　大島聡史 [2]　片桐 孝洋 [2]　横田達也 [3]　本谷秀堅 [3]　永井亨 [2]

本発表では，統計形状モデルを構築するため 2 つの臓器間の微分同相写像を求める LDDMM コードを，複数の GPU を用いて並列化し実行時間の短縮を行った．東京大学設置の Reedbush-H スーパーコンピュータシステムと名古屋大学情報基盤センター設置の GPU サーバ sx40 を用いて性能評価を行った．性能評価の結果から，オリジナルの CPU 逐次プログラム実行に対して，Reedbush-H（1 ノード，2GPU）では最大 188.44 倍，sx40（1 ノード，4GPU）では最大 319.56 倍の高速化を達成した

1. はじめに

MRI や X 線 CT 画像といった医用画像を用いた診断に際して重要な指標の一つは各臓器の形である．臓器機能の変化はその臓器の形態に影響を及ぼすことがある．例えば肝硬変は肝臓を変形させることが多い [1]. また, 脳機能の低下と脳の特定部位の委縮に強い相関があることが知られている [2]．このような正常でない臓器形状を検知する上で，正常な患者の統計形状モデルを構築することは重要である．

臓器の統計形状モデルは学習用 3 次元画像中の臓器領域データ群より構築される．臓器の統計形状モデルを求める手法のうち最も広く知られている手法は Active Shape Model（ASM） [3]である．ASM は形状の多様性を線形にモデル化する手法であるが，構築されたモデルは表現性能[4]が十分でない．例えば肝臓や腎臓のような単純な（自己交差しない）閉曲面で表現できる臓器について，学習データがすべて単純な閉曲面であったとしてもそれらによって構築された線形モデルが表現する形状には自己交差する局面，すなわち臓器として適切ではない形状が含まれる可能性がある．そのため妥当な形状のみを表現するモデルの構築方法として臓器領域間の非線形な微分同相写像（1 対 1 かつ順写像も逆写像も滑らかな写像）によるモデル化を用いるのが良い．

微分同相写像を用いて統計形状モデルを構築するためには，まず学習用の臓器ボリュームデータの平均を求め，その平均形状から各学習データへの微分同相写像をもとめる．この際求める平均形状の算出には，単なる臓器領域の加算平均では同相写像と整合しなくなるため，学習データ間の微分同相写像を繰り返し計算する必要がある [5].

2 つの臓器間の微分同相写像を求める手法の中でも Large Deformation Diffeomorphic Metric Mapping（LDDMM）[6]と呼ばれる手法が広く採用されている．LDDMM によって求められる微分同相写像は正則化項を含むコスト関数の繰り返し計算により最適化して得られる写像であり，多くの計算を必要とする．それにも関わらず統計形状モデルを構築するためにはこの手法を多数の学習データに対して多数回適用しなければならない．そのため臨床で用いる医用画像と同程度の高い空間分解能で高精度な統計形状モデルを構築するためには LDDMM の高速化が必要不可欠である．

中島ら [7]は LDDMM コードをマルチコア CPU 向けにハイブリッド MPI/OpenMP 並列化し高速化したが，その実行時間のうちの多くは浮動小数点演算で占められていることが分かっている．したがって，高い浮動小数点演算性能を持つ Graphics Processing Unit（GPU）を活用することでさらなる高速化が期待される．

GPU はもともと画像のレンダリングのために利用されてきたが，過去十数年でその計算資源を数値計算などのより広い目的で使う動きが High-Performance Computing（HPC）分野を中心に広がり, 医用画像処理分野においても GPU による並列計算が広く活用されるようになってきた [8]．しかしそれらの研究の多くは 1 枚の GPU のみを用いており，GPU を複数枚用いて並列に計算させる（マルチ GPU）ことによってさらなる高速化が期待できる．そのため，医用画像処理分野においてマルチ GPU の活用を推進していくことは重要な課題である．

以上から本研究ではハイブリッド MPI/OpenMP 並列化された LDDMM コードをベースとして，複数の GPU を用いた LDDMM コードを実装し性能評価を行う．

本稿は以下の構成からなる．まず第 2 章で LDDMM とその実装に関する説明を行う．次に第 3 章では GPU を用いて実行するための手法とその具体的な実装を示し，第 4 章で複数の環境・実装方法における性能評価と比較を行う．最後に第 5 章では本稿のまとめを行い，これからの課題について述べる．

1 名古屋大学大学院情報学研究科
Graduate School of Informatics, Nagoya University
2 名古屋大学 情報基盤センター
Information Technology Center, Nagoya University
3 名古屋工業大学大学院 工学研究科
Graduate School of Engineering, Nagoya Institute of Technology,

ⓒ2020 Information Processing Society of Japan

2. LDDMM アルゴリズム及びコード

本研究で利用する LDDMM コードは，LDDMM に基づき微分同相写像が導き出され，最急降下法による最適なベクトル場の推定が利用されている．この章では LDDMM アルゴリズムの説明を行う．

2.1 LDDMM

LDDMM は 2 枚の入力画像I_0, I_1に対して，I_0からI_1への微分同相写像の生成を行う．ここで，I_0をテンプレート画像，I_1をターゲット画像と呼ぶこととする．画像が定義される領域を$\Omega \in R^n$（n は領域の次元数）で定義すると，テンプレート画像およびターゲット画像の画素値をそれぞれ$I_0 : \Omega \to \mathbb{R}^d$，$I_1 : \Omega \to \mathbb{R}^d$（入力画像がグレースケールなら$d = 1$，カラーイメージなら$d = 3$とする）で表され，求める微分同相写像は$\phi : \Omega \to \Omega$と表される．$t \in [0,1]$として，$\phi_0$は恒等写像，$\phi_1 = \phi$を満たす時間$t$後の写像$\phi_t$を求めることによって微分同相写像$\phi$を求めることができる．

テンプレート画像I_0のある座標を$\boldsymbol{x}_0$としてϕ_tによる時間tの変形後の座標を$\boldsymbol{x}_t$とすると，以下の式が成立する．

$$\boldsymbol{x}_t = \phi_t(\boldsymbol{x}_0)$$

…(1)

この式が示すように，2 枚の入力画像から写像を求めることで新たなモデル構築が可能となる．肝臓平均形状I_0をテンプレート画像，肝臓症例I_1をターゲット画像として入力したときの LDDMM 例を図 **1** に示す．図が示すように 2 つの入力画像からその中間の新たな 3 次元画像を得ることができている．

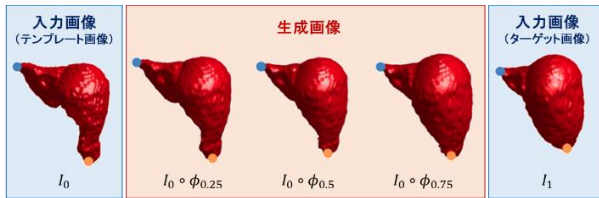

図 1　肝臓平均形状から肝臓症例への LDDMM

写像ϕ_tは時間変化するベクトル場$v_t : \Omega \to R^n$を用いて以下の式ように表される．なお，v_tはϕ_tの時間微分である．

$$\phi_t = \int_0^t \boldsymbol{v}_t dt + \phi_0$$

…(2)

式(2)から，写像ϕ_tはベクトル場v_tを求めることにより導出される．v_tは以下の式によって求められる．

$$\hat{\boldsymbol{v}}_t = \underset{\boldsymbol{v}_t}{argmin}\left(\int_0^1 \|\boldsymbol{v}_t\|_V^2 dt + \frac{1}{\sigma^2}\|I_0 \circ \phi_1^{-1} - I_1\|_{L^2}^2\right)$$

…(3)

式(3)における$\frac{1}{\sigma^2}\|I_0 \circ \phi_1^{-1} - I_1\|_{L^2}^2$はテンプレート画像の変形後の画素値とターゲット画像の画素値との二乗誤差を表している．また，式(3)の$\int_0^1 \|v_t\|_V^2 dt$は$\boldsymbol{v}_t$を滑らかにするために導入される正則化項である．$\boldsymbol{\phi}_t$を微分同相な写像にするためには$\boldsymbol{v}_t$を十分滑らかなベクトル場に限定すればよいことが知られており，ベクトル場のノルム$\|\cdot\|_V^2$を次のように定義することで$\boldsymbol{v}_t$は十分滑らかとなることが知られている[6].

$$\langle \boldsymbol{f}, \boldsymbol{g} \rangle_V = \langle L\boldsymbol{f}, L\boldsymbol{g} \rangle_{L^2} = \langle L^\dagger L\boldsymbol{f}, \boldsymbol{g} \rangle_{L^2}$$

…(4)

ただし，$\boldsymbol{f}, \boldsymbol{g}$はそれぞれベクトル場であり，$L$は微分作用素で$L \doteq -\alpha\nabla^2 + \gamma$，$L^\dagger L$は$L$の随伴行列である．これにより写像$\phi_t$が微分同相写像となるための条件を満たす．

I_0からI_1への変形ベクトル$\hat{\boldsymbol{v}}$を以下のように表す．

$$\hat{\boldsymbol{v}} = \underset{\boldsymbol{v}}{arginf}\, E(\boldsymbol{v})$$

$$E(\boldsymbol{v}) \doteq \int_0^1 \|\boldsymbol{v}_t\|_V^2\, dt + \frac{1}{\sigma^2}\left\|I_0 \circ \phi_{1,0}^v - I_1\right\|_{L^2}^2$$

…(5)

式(5)に示す最小化問題の解は以下の Euler-Lagrange 方程式を満たす．

$$2\hat{\boldsymbol{v}}_t - K\left(\frac{2}{\sigma^2}\ |D\phi_{t,1}^{\hat{v}}\,|\nabla J_t^0(J_t^0 - J_t^1)\right) = 0$$

…(6)

ただし$\phi_{s,t} = \phi_t \circ (\phi_s)^{-1}$であり，$\mathrm{J}_\mathrm{t}^0 \doteq I_0 \circ \phi_{t,0}$, $J_t^1 \doteq I_1 \circ \phi_{t,1}$とする．$K : L^2(\Omega, \mathbb{R}^d) \to V$はコンパクト自己随伴作用素であり，$\langle a, b \rangle_{L^2} = \langle Ka, b \rangle_V$のように定義する．また，エネルギー関数の勾配は以下のように表される．

$$(\nabla_v E_t)_V = 2\hat{\boldsymbol{v}}_t - K\left(\frac{2}{\sigma^2}\ |D\phi_{t,1}^{\hat{v}}\,|\nabla J_t^0(J_t^0 - J_t^1)\right)$$

…(7)

LDDMM コードではエネルギー関数の勾配を用いた再急降下法によりエネルギー関数が最適となる$\boldsymbol{v}$を求める．$t_j \in [0, T], j \in \{0,1,2,\ldots,N\}$とし，$\delta t = \frac{T}{N}$ とすると，

$$\boldsymbol{v}^{k+1} = \boldsymbol{v}^k - \epsilon\nabla_{\boldsymbol{v}_{t_j}^k} E$$

…(8)

ベクトル場$\boldsymbol{v}_{t_j}^k(y)$，写像$\phi_{t_j}^k(y)$を，$k$回アルゴリズムを繰り返し$j$番目の時間で得られたものとすると，式(7)に示したエネルギー関数の勾配を離散化すると以下のようになる．

$$\nabla_{\boldsymbol{v}_{t_j}^k} E_{t_j}^k(y)$$

$$= 2\boldsymbol{v}_{t_j}^k(y) - \frac{2}{\sigma^2}K\left(\left|D\phi_{t_j,T}^k(\boldsymbol{y})\right| DJ_{t_j}^0(y) * \left(J_{t_j}^0(y) - J_{t_j}^T(y)\right)\right)$$

…(9)

また，式(5)に示したエネルギー関数は以下のように表される．ただしI_0のサイズを$N_1 \times N_2 \times N_3$とする．

$$E(\boldsymbol{v}^k) = \sum_{j=0}^{N-1}\left\|\boldsymbol{v}_{t_j}^k\right\|_V^2 \delta t + \frac{1}{N_1 N_2 N_3} \times \sum_{y \in \Omega}\left|J_T^0(y) - J_T^T(y)\right|^2$$

ⓒ2020 Information Processing Society of Japan

2.2 LDDMM コード

微分同相写像の計算部分は以下の 4 ステップを繰り返す構成となっている．

Step1. ヤコビアンの演算

Step2. Backward Integration

Step3. ベクトル場 v の更新

Step4. 写像 ϕ と位置情報の更新

微分同相写像を得るには Step1 から Step4 を妥当な写像を得ることができるまで繰り返せばよいが，本研究ではこの反復回数 G に一定の値を与えて実験を行う．各ステップは複数の多重ループにより構成されており，多くの演算を含んでいる．

本研究では LDDMM コードの MPI と OpenMP による並列化（ハイブリッド MPI/OpenMP）実装によりマルチコア CPU 向けに並列化されたコードをベースとして GPU による高速化を試みる．ベースとなるコードでは写像計算部分のうち逐次実行時に最も時間のかかる Step3 に焦点を当てて高速化が行われている．具体的には，各ステップに含まれている for ループを OpenMP でスレッド並列化を行い，特に時間のかかる Step3 に含まれる多重ループについてはプロセスごとにループを分割して実行し MPI_Allgather 通信関数によって各プロセスのデータを集約している．

3. GPU を用いた並列化

本研究ではマルチコア CPU 向けに実装された LDDMM コードをベースとして，OpenACC [9]と GPU 間通信ライブラリを用いてマルチ GPU 向けの実装を行った．この章ではその詳細について記述する．

3.1 OpenACC による並列化

GPU を用いた場合の LDDMM コードの流れを図 2 に示す．図中の黒字の部分は CPU のみで処理されるが，緑字の部分は OpenACC のディレクティブにより指示される GPU を用いた処理である．本研究では OpenACC ディレクティブを用いて，微分同相写像計算部分を構成している全ての for ループに対して図 3 のように GPU へのオフロードを指示している．本研究で利用したコードはすでに for ループに対して OpenMP ディレクティブによりスレッド並列化がされているため，それを OpenACC ディレクティブに書き換えるような形で容易に GPU 化することができた．

3.2 マルチ GPU を用いた並列化

本研究でベースとしたマルチコア CPU 向けの LDDMM コードでは写像計算部分において MPI によるプロセス間通信を用いた並列処理が実装されている．本研究ではこれを利用して複数の GPU を用いた並列処理を実装する．MPI によって立ち上げたプロセスをそれぞれ GPU に割り当て，GPU 間通信ライブラリを用いてマルチ GPU 並列処理を行う．

GPU 間通信のライブラリとしては GPU 間通信ライブラリとして広く利用されている CUDA-Aware MPI を用いた．また，本研究で扱う通信は主に GPU デバイスメモリ間のみの集団通信であることから，マルチ GPU 集団通信に最適化されている GPU 間通信ライブラリ NVIDIA Collective Communications Library (NCCL) [10]を利用した場合についても性能評価を行った．

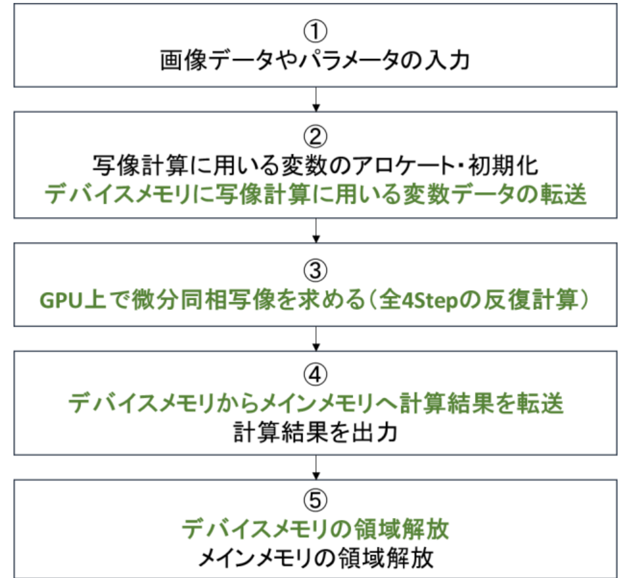

図 2　GPU を用いた LDDMM コードの流れ

```
#pragma acc kernels loop independent gang
collapse(2)
for(i=xorder[n];i<=xorder[n+1];i++) {
    for(j=0; j<y; j++) {
        #pragma acc loop independent vector
        for(k=0; k<z; k++) {
            ～演算～
        }
    }
}
```

図 3　OpenACC によるオフロード指示の例

4. 実験

4.1 問題設定

問題サイズを 100×100×100 に固定し，Step1 から Step4 の処理の反復回数を 1500 回に固定して性能評価を行った．時間計測は写像計算部分のみについて行い，その前後の配列データの確保・解放や転送の部分は計測範囲に含めない．

4.2 実験環境

東京大学情報基盤センター設置の Reedbush-H [11]，名古屋大学情報基盤センターで試験運用されている GPU サーバ sx40 [12]を用いてそれぞれ性能評価を行った．Reedbush-

ⓒ2020 Information Processing Society of Japan

H の計算機構成を，**表 1** に示す．また，sx40 の計算機構成を**表 2** に，ハードウェア構成図を**図 4** に示す．

表 1　Reedbush-H の構成

Reedbush-H のハードウェア構成		
CPU		Intel Xeon E5-2695 ×2
GPU		NVIDIA Tesla P100 × 2（GPU 間接続：NVLink）
メインメモリ		256 GB
ソフトウェア構成		
コンパイラ		PGI 19.10 C++コンパイラ
コンパイルオプション	CPU のみの場合	-std=c++11 -O3 -fast
	GPU を用いる場合	-std=c++11 -O3 -fast -Mcuda -acc -lnccl -ta=tesla:cc60
ライブラリ	MPI（CUDA-Aware MPI）	MVAPICH2 2.3.3
	OpenACC	OpenACC 2.6（PGI 19.10）
	CUDA	CUDA 10.0
	NCCL	NCCL 2.4.2

表 2　sx40 の構成

sx40 のハードウェア構成		
CPU		Intel Xeon Gold 5122 × 2
GPU		NVIDIA Tesla V100 × 4（GPU 間接続：NVLink2）
メインメモリ		384 GB
ソフトウェア構成		
コンパイラ		PGI 19.10 C++コンパイラ
コンパイルオプション	CPU のみの場合	-std=c++11 -O3 -fast -tp=Skylake
	GPU を用いる場合	-std=c++11 -O3 -fast -Mcuda -acc -lnccl -ta=tesla:cc70
ライブラリ	MPI（CUDA-Aware MPI）	OpenMPI 3.1.3（PGI 19.10）
	OpenACC	OpenACC 2.6（PGI 19.10）
	CUDA	CUDA 9.2
	NCCL	NCCL 2.2.13

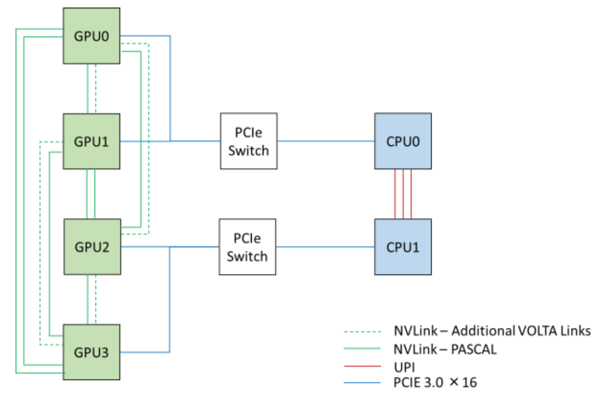

図 4　GPU サーバ sx40 のブロックダイアグラム

4.3 Reedbush-H 上での実験結果

4.3.1 CPU 逐次実行と GPU1 枚を用いた場合の実行時間

CPU 逐次実行時間は 4,774,534.31(ms)であった．これに対して GPU1 枚を用いた場合の実行時間は 30,427.59(ms)であり，CPU 逐次実行に対して 156.91 倍高速となった．

4.3.2 複数の GPU を用いた場合の実行時間

次に， GPU を二枚用いた場合について，通信ライブラリとして MVAPICH2 を用いた場合と NCCL を用いた場合それぞれの実行時間および GPU を 1 枚用いた場合との速度比を**表 3** に示す．また，それぞれの実行時間のうち通信時間と演算時間の内訳を示すグラフを**図 5** に示す．NCCL を用いた場合は MVAPICH2 を用いた場合よりも通信時間が短いため全体の実行時間が最も短くなっており，1 枚の GPU を用いた場合に対して 1.20 倍高速となった．また，これは CPU 逐次実行に対して 188.44 倍高速であった．

表 3　Reedbush-H を用いた実行時間

	1GPU	2GPU (MVAPICH2)	2GPU (NCCL)
実行時間(ms)	30427.59	27217.65	25336.54
速度比		1.12	1.20

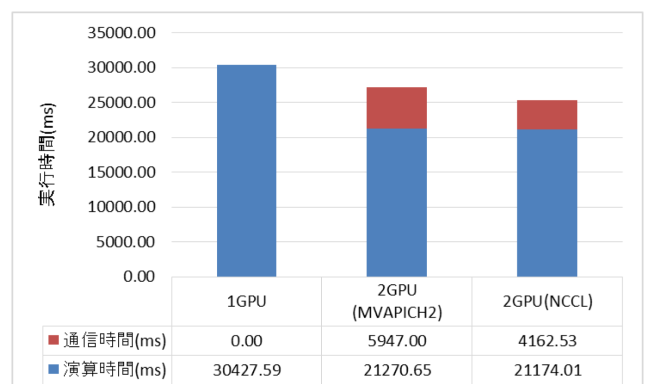

図 5　実行時間のうちの通信時間と演算時間（Reedbush-H）

ⓒ2020 Information Processing Society of Japan

4.4 sx40 上での実験結果

4.4.1 CPU 逐次実行と GPU1 枚を用いた場合の実行時間

CPU 逐次実行時間は 3,126,985.01 (ms)であった．これに対して GPU1 枚を用いた場合の実行時間は 13,553.55(ms)であり，CPU 逐次実行に対して約 230.7 倍高速となった．また，Reedbush-H を用いた場合の 1GPU 実行と比較すると，2.24 倍高速であった．

4.4.2 Open MPI を用いた場合の実行時間

次に，Open MPI を用いて GPU の枚数を 1 枚から 4 枚まで変化させたときの実行時間を**表 4** に示す．また，それぞれの実行時間のうち通信時間と演算時間の内訳を示すグラフを**図 6** に示す. 1GPU の時が最も速いという結果となり，複数の GPU を用いることによってさらに高速とはならなかった．演算時間と通信時間の内訳に注目すると，複数の GPU を用いる場合は通信時間に大きく時間がかかってしまい，演算時間の減少に対して大きくなりすぎていることで，全体として複数の GPU を用いる場合の実行時間が大きくなってしまっていることが分かる．

表 4　sx40 を用いた実行時間（Open MPI）

	1GPU	2GPU	3GPU	4GPU
実行時間 (ms)	13553.55	17714.95	17095.76	15157.05
速度比		0.765	0.793	0.894

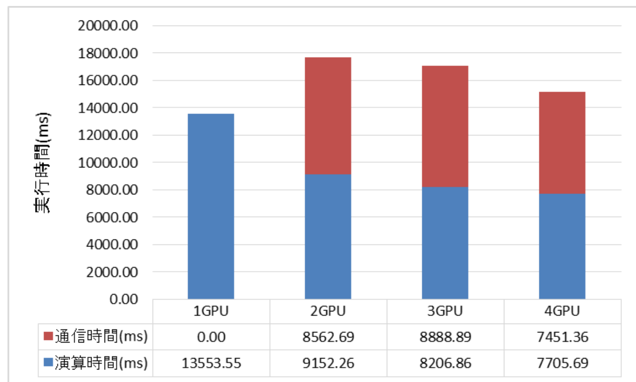

図 6　実行時間のうちの通信時間と演算時間（sx40，Open MPI）

4.4.3 NCCL を用いた場合の実行時間

次に，NCCL を用いた場合の実行時間を**表 5** に示す．また，実行時間のうち通信時間と演算時間の内訳を表すグラフを**図 7** に示す. Open MPI を用いた場合に比べて通信時間が大幅に削減されていることが分かる．これにより複数の GPU を用いた場合の実行時間も削減されており，全体として 4GPU の場合が最も高速となり，1GPU に対して 1.39 倍，CPU 逐次実行に対しては 319.56 倍高速となった．

表 5　sx40 を用いた実行時間（NCCL）

	1GPU	2GPU	3GPU	4GPU
実行時間 (ms)	13553.55	12558.06	10655.92	9785.36
速度比		1.08	1.27	1.39

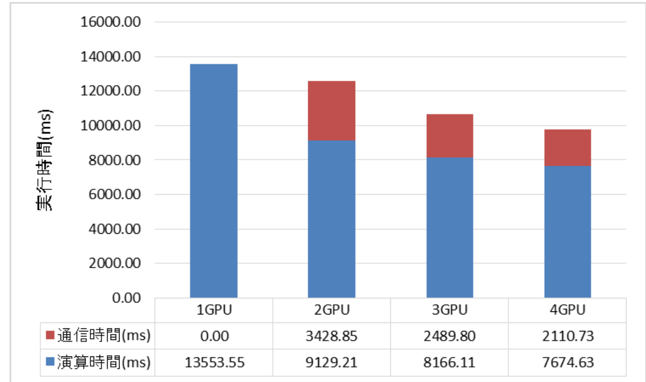

図 7　実行時間のうちの通信時間と演算時間（sx40，NCCL）

4.4.4 演算時間の高速化比率について

4.4.3 で計測した実行時間のうちの演算時間について，ステップごとの内訳を**表 6** に示す．演算時間に関して，4GPU は 1GPU に対して Step3 では 2.47 倍であるが Step4 は全く速くなっておらず，結果として GPU を 4 枚まで増やしても演算時間合計は 1.77 倍程度にしか高速化されていない．これは本研究でベースとした MPI 実装において Step4 が MPI 並列化されていないためである．しかし，本アプリケーションでは入力パラメータを大きくしていくと Step4 に対して Step3 の計算量の比率が大きくなることが分かっている．本研究においてはベンチマークとして比較的小さいパラメータを入力として実験を行っているが，より大規模な入力の計算を行う際には Step3 の演算時間の比率が大きくなっていくため，本実験で得られた 1.77 倍の速度比よりもさらに高速になると考えられる．

表 6　ステップごとの演算時間

	1GPU	4GPU	速度比
Step1 演算時間 (ms)	61.02	65.27	0.93
Step2 演算時間 (ms)	136.69	122.20	1.12
Step3 演算時間 (ms)	9862.92	3999.46	2.47
Step4 演算時間 (ms)	3492.93	3487.70	1.00
演算時間合計 (ms)	13553.55	7674.63	1.77

ⓒ2020 Information Processing Society of Japan

4.4.5 GPU 間通信の通信量

パフォーマンスカウンタを用いて LDDMM コードの実行前後の NVLink の総送信（受信）量の差をとることで，実行中の NVLink を介したデータの送受信量を測定した．Open MPI を用いて GPU 間通信を行った場合の NVLink の通信量 (受信量と送信量) を測定した結果をそれぞれ**図 8**, **図 9** に示す．また，NCCL を用いて GPU 間通信を行った場合の NVLink の通信量（受信量と送信量）を測定した結果をそれぞれ**図 10**, **図 11** に示す．Open MPI を用いた場合は送受信ともに Link ごとのデータ量の差が大きく，1 つの Link での最大の通信量は約 45GB にも達していることが分かる．それに対して NCCL を用いた場合は送受信ともにデータ量は Link ごとにほぼ均等であり，1 つの Link での最大通信量は約 22.5GB であった．このことから NCCL による通信は OpenMPI に比べて GPU のトポロジを活用した効率の良い集団通信を行うことができていると考えられる．

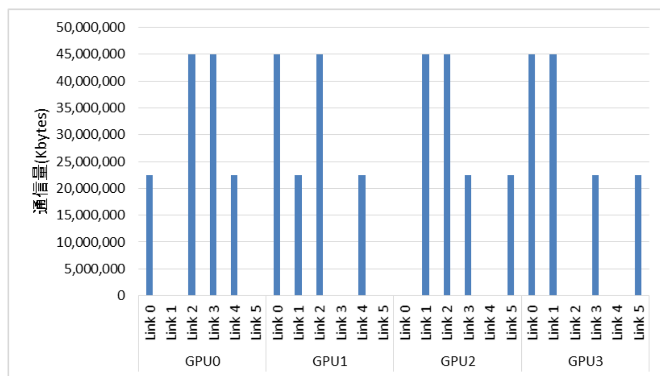

図 8　NVLink の受信量(Open MPI)

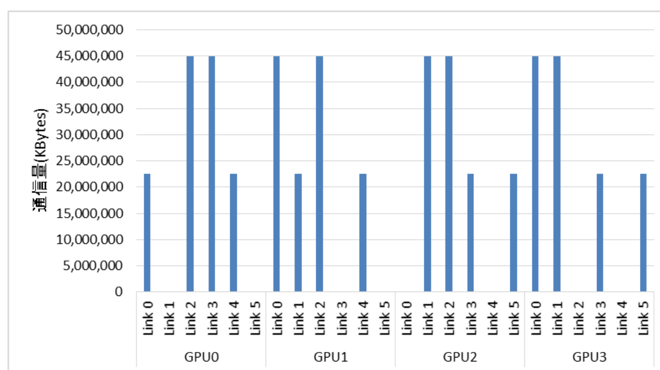

図 9　NVLink の送信量(Open MPI)

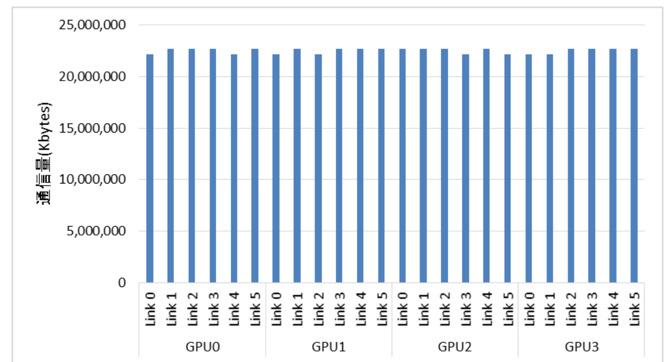

図 10　NVLink の受信量(NCCL)

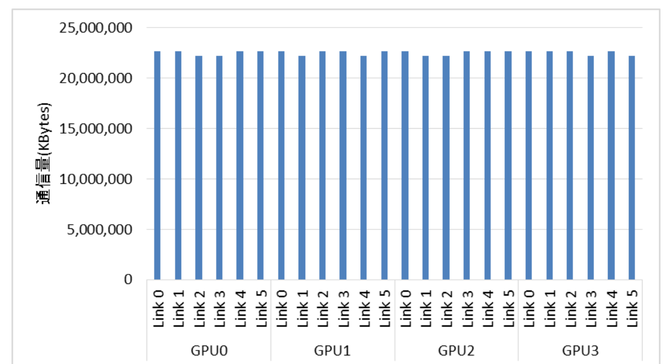

図 11　NVLink の送信量(NCCL)

5. おわりに

本研究では LDDMM コードの GPU による高速化を試みた．OpenACC による GPU 並列化と CUDA-Aware MPI や NCCL を用いた GPU 間通信とを組み合わせてそれぞれ性能評価を行い，GPU 化による LDDMM コードの大幅な高速化を達成した．

東京大学情報基盤センター設置の Reedbush-H を用いた実験では GPU を 2 枚用いて GPU 間通信に NCCL を用いた場合が最も速く，CPU を用いた逐次実行と比較して 188.44 倍となった．名古屋大学情報基盤センター設置の GPU サーバ sx40 を用いた実験では GPU を 4 枚用いて GPU 間通信に NCCL を用いた場合が最も速く，319.56 倍の高速化を達成した．また, NVLink の通信量の計測結果から OpenMPI を用いた GPU 間通信と比較して NCCL の方が効率的なデータ転送を行えておりアプリケーションの高速化に有効であることが確認できた．

本研究を通じて，医用画像処理分野において既存コードの OpenACC による GPU 化の有効性とマルチ GPU による並列計算の有用性を示すことができた．

今回は 1 ノードで比較的小さいパラメータによる実験を行ったが，複数ノードを用いた大規模な環境においての実行・評価は今後の重要な課題である．

ⓒ2020 Information Processing Society of Japan

謝辞 本研究は JSPS 科研費 JP18H03262 ，学際大規模情報基盤共同利用・共同研究拠点(課題番号: jh200046-DAH)の助成を受けたものです

参考文献

[1] 小原 伸哉，Amir Hossein Foruzan，健山 智子 他，“肝臓の統計的形状モデル構築と肝硬変診断支援への応用,” 電子情報通信学会技術研究報告, 2010.

[2] Ph. Scheltens, F. Barkhof, D. Leys, E. Ch. Wolters, R. Ravid, W. Kamphorst, “Histopathologic correlates of white matter changes on MRI in Alzheimer's disease and normal aging,” Neurology, 1995.

[3] T.F. Cootes, C.J. Taylor, D.H. Cooper, J. Graham, “Active Shape Models-Their Training and Application,” Computer Vision and Image Understanding, Volume 61, Issue 1, Pages 38-59, 1995.

[4] Styner M.A. et al., “Evaluation of 3D Correspondence Methods for Model Building,” Information Processing in Medical Imaging, 2003.

[5] John Ashburner, Karl J. Friston, “Computing average shaped tissue probability templates,” NeuroImage, Volume 45, Issue 2, Pages 333-341, 2009.

[6] M.F.Beg, M.Miller, A.Trouve, and L.Younes, “Computing metrics via geodesics on flows of diffeomorphisms,” Internation Journal of Computer Vision, vol.61, no.2, pp.139-157, 2005.

[7] 中島 大地，田中 友揮，物部 峻太郎，本谷 秀堅，横田 達也，片桐 孝洋，永井 亨，荻野 正雄，“医用画像処理における LDDMM の並列化とコード最適化,” 情報処理学会研究報告 pp.1-7, 2018.

[8] T. Kalaiselvi, P. Sriramakrishnan, K. Somasundaram, “Survey of using GPU CUDA programming model in medical image analysis,” Informatics in Medicine Unlocked, Volume 9, Pages 133-144, 2017.

[9] OpenACC, [オンライン]. Available: https://www.openacc.org/.

[10] NVIDIA Collective Communications Library(NCCL), [オンライン]. Available: https://developer.nvidia.com/nccl.

[11] 東京大学基盤センタースーパーコンピューティング部門，“Reedbush-H,” [オンライン]. Available: https://www.cc.u-tokyo.ac.jp/guide/hpc/rbh/.

[12] 名古屋大学情報連携統括本部 - GPU サーバの試験運用について，[オンライン]. Available: http://www.icts.nagoya-u.ac.jp/ja/sc/news/maintenance/gpu.html.

ⓒ2020 Information Processing Society of Japan