Vol.2022-HPC-183 No.22
2022/3/18

グラフニューラルネットワークによる長時間分子動力学予測と性能評価

芝　隼人[1,a)]　下川辺　隆史[1,b)]

概要：分子動力学シミュレーションは，所与の物理系の構成要素である原子分子をその運動方程式に従って運動させることによって物質の性質を予言する強力な手法としての位置を確立している．しかし，長時間の分子動力学の時間積分にについては演算器のクロックサイクルによってその速度が制限され，１日あたり 10^{10} ステップ数を超えるような長時間の計算は今後も困難である可能性が高い．最近，グラフニューラルネットワークが何万ステップ以上を隔てたステップ数の分子動力学の結果を良く予測することが見出された．この予測手法を用いることで分子動力学シミュレーションの適用対象を飛躍的に広げられる潜在的可能性を持っている．本発表では，このアプローチに対する我々の現在の取り組みを紹介するとともに，GPU (Graphics Processing Units) におけるグラフニューラルネットワークによる学習性能の評価を実施する．

Graph Neural Networks Prediction of Long-Time Molecular Dynamics and its Benchmarks

Hayato Shiba[1,a)]　Takashi Shimokawabe[1,b)]

1. はじめに

近年隆盛をきわめているデータ科学は，実験・観測データを扱う様々な分野に変革をもたらしている．この中で実験データを直接扱わない科学技術シミュレーションそのものについても，データ科学手法によって加速しようとする研究も活発になっている．本稿の第一著者が専門とする物質・材料系の分子動力学シミュレーションそのものに話を限定すると深層学習を用いたシミュレーションの高度化には，近年２つの主な大きな流れがあるように思われる．一つ目は，量子力学に基づいた高精度なシミュレーションの結果をニューラルネットワークに学習させ，分子動力学の力場パラメターに反映させこれを共有する「機械学習分子動力学」という一連の手法であり，量子力学レベルの精度での多体分子動力学を一挙に可能にした．最近大きな普及を見せた [1], [2], [3]．2020 年には，100 万原子系の高精度な計算が SC20 Gordon Bell 賞を受賞するに至った [4]．もう一つの大きな，より以前からある研究の流れとして，単純な数値時間積分では取り扱えない長時間現象を扱うための分子動力学の手法開発がある．蛋白質の構造変化などに代表されるミリ秒以上の長時間プロセスは，古典分子動力学でも扱える時間スケール（全原子力場では 10^{10} オーダーのステップ数）を大きく超え．そのダイナミクスを扱うには時間方向への何らかの加速が必要である．所与の物理系の自由エネルギー地形の谷底間の遷移を捉える適切な座標系を抽出できれば，その方向にバイアスをかけたシミュレーションを行うか [5]，あるいはその経路上での時空軌道のサンプリング [6] が行うことができるとされ，研究が行われてきた．ただし，これらの手法の利用する際には，軌道を記述するのにふさわしい限定された数の「適切な集団座標」を事前に同定することが要求される．一般論としては，複雑な自由エネルギー地形内で状態遷移が起こる場合に，しばしばこれらの手法の利用は困難となる．

[1] 東京大学情報基盤センター
Information Technology Center, University of Tokyo, Chiba 277-0882, Japan
[a)] shiba@cc.u-tokyo.ac.jp
[b)] shimokawabe@cc.u-tokyo.ac.jp

我々は，分子動力学シミュレーションについて，長時間の現象を取り扱うための全く違った方向性を見出すべく，検討を行っている．この中で，2020 年に，DeepMind および Google Brain に所属する 11 名からなる研究グループによって，ガラスと呼ばれる物質のシミュレーションをグラフニューラルネットワーク (GNN) による機械学習によって大変よく予測できるとする研究が発表された [7]．ガラスは液体を急冷すると形成される，非常に遅く緩和する液体（いわば「固まった液体」）の総称であり，普遍的に存在する物質の状態を指す．結晶とは異なって原子・分子の配置に周期性がないため現象の把握が非常に困難となっており，さらに知りたい性質は動力学上のものであるため多くの熱統計力学の知識が通用しない．遅く緩和する以上は内部の原子分子はなかなか入れ替わらないものの，その入れ替えは熱ゆらぎに伴って確率的に発生するため，分子動力学シミュレーションのステップを省略してくれるような有力な手法が存在しないのが現状である．

文献 [7] では何十万ステップという長時間後にガラスの内部でどの原子がどの程度動くのかを学習すると，**最初の粒子配置についての情報だけから**予測が可能になることが主張されている．粒子の長時間にわたる運動性の情報性が１枚の粒子配置の情報に埋め込まれているという考え方は以前より分野で共有されてきた [8] ものである．しかし，今回 GNN がこの考え方に基づく予言や最近提案された機械学習手法 [9], [10] よりも定量的に正確な予言を与えることから，分野で大きく注目されるに至っている．我々は，この手法が分子動力学シミュレーションの長時間予測に対して大きな可能性を持っているものと考え，研究テーマとして取り上げることにした．現在我々は，より包括的な時間範囲をカバーするデータセットを我々自身で取得した上に立って，先行研究 [7] において学習対象となる粒子の動きやすさ（以降，「易動度（propensity）」と呼ぶ）より物理的に予測を改善するような特徴量を同定するための研究を行っている．本稿では，その準備として行った GNN による学習の実装，また性能測定評価を実施した結果を報告する．

2. データセット作成

2.1 分子動力学モデルと計算条件

本研究では文献 [7] に準じた形でデータセットを作成しており，性能評価に使用したデータセットの作成方法について本節にて述べる．データセットは，著者自身によって作成されたスクラッチコードによる分子動力学の実行結果から得られた，構成粒子が運動する前と運動した後の位置情報について記述したものである．シミュレーションにおいては先述した通り液体を急冷することによって過冷却状態を実現しているが，ガラス転移点には到達していないため完全に凍結されずにゆっくりと緩和する．この状態での結晶化が妨げられるように，２つの種類の粒子を混合した３次元 Kob–Andersen Leannard–Jones (KALJ) 液体 [11] という古くからガラスの標準モデルとして使用されている相互作用モデルを用いる．ただし，オリジナルのモデルからカットオフ距離点におけるポテンシャルおよび力の連続性を保証するためスムージング [12], [13] をかけた次の修正ポテンシャルを使用した．

$$V(r) = u(r) - u(r_c) - (r - r_c) \left.\frac{du(r)}{dr}\right|_{r=r_c} \tag{1}$$

$$u(r) = 4\epsilon_{\alpha\beta}\left[\left(\frac{\sigma_{\alpha\beta}}{r}\right)^{12} - \left(\frac{\sigma_{\alpha\beta}}{r}\right)^{6}\right] \tag{2}$$

ここに，α, $\beta \in \{\mathrm{A,B}\}$ は粒子の種別を表すインデクスである．本研究では全粒子数を $N = 4096$ としており，80% (3277 粒子) が A，20% (819 粒子) が B の種類とする．２粒子間相互作用において粒子径 $\sigma_{\alpha\beta}$ および相互作用エネルギー $\epsilon_{\alpha\beta}$ が種類の組ごとに異なっており，前者は $\sigma_{\mathrm{AB}} = 0.8\sigma_{\mathrm{AA}}$，$\sigma_{\mathrm{BB}} = 0.88\sigma_{\mathrm{AA}}$，後者は $\epsilon_{\mathrm{AB}} = 0.5\epsilon_{\mathrm{AA}}$，$\epsilon_{\mathrm{BB}} = 1.5\epsilon_{\mathrm{AA}}$ と設定するが，質量は全ての粒子が等しく m としている．相互作用のカットオフ距離は種類の組ごとに $r_c = 2.5\sigma_{\alpha\beta}$ とする．以降，特に断りのない限り長さ，温度，時間は σ_{AA}，ϵ/k_{B}，$\sigma_{\mathrm{AA}}\sqrt{m/\epsilon_{\mathrm{AA}}}$ をそれぞれ単位として記載する（k_{B} はボルツマン定数）．

シミュレーションは数密度が $N/V = 1.2$ となるように設定された体積 V の立方体領域にて一定体積条件で実施し，差分ステップは $dt = 10^{-3}$ に固定する．周期境界条件のもとでランダムに粒子を配置した系を高温 $T = 5.0$ で 10^7 ステップにわたって攪拌して液体状態を作成した後，10^4 ステップという短時間で目的温度（本原稿では $T = 0.47$）まで急冷する．そして引き続き，能勢–フーバー熱浴 [14] を使用して緩和時間の 40 倍以上に相当する 2.5×10^7 ステップの長時間にわたって待機する．このようにしたあと，粒子が乱雑な配置に固まったまま状態が安定した適当なタイミングを時間起点 ($t = 0$) における配置のデータとして保存する．これらはガラスのシミュレーションを行う上で標準的に使用されてきた方法に基いている [15]．以上を目的温度に対して異なる 100 パターンの初期条件に対して行ったあと，これを利用したデータセット生成に入る．図 1 には密度緩和を表現する中間散乱関数 $F_{\mathrm{s}}(k = 2\pi/\sigma_{\mathrm{AA}}, t)$（$k$ は波数を表す）を，他の温度も含めて評価した結果を示す．シミュレーション中においては，力の計算についても時間積分についても，全て FP64 精度で計算している．次節に述べる手順に従って粒子の易動度（propensity）のデータを取得し，GNN での学習に利用するデータセットとする．

2.2 同一配座アンサンブル

データセットの作成は文献 [7] に倣う形で「同一配座アンサンブル (isoconfigurational ensemble)」 [16] を取得する．この手法においては，$t = 0$ における粒子配置１パターンに

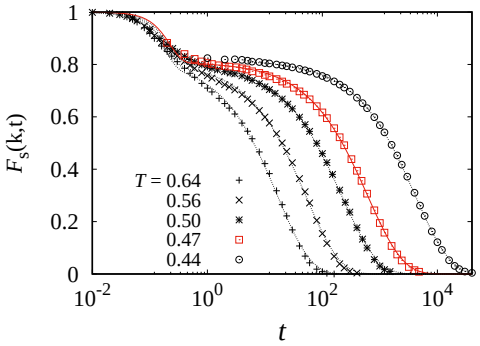

図 1 データセット作成に用いた 3 次元 Kob-Andersen Lennard-Jones 液体の密度場の緩和を表す中間散乱関数 $F_s(k,t)$ を 5 通りの温度条件に対して測定しプロットした．この関数は 1 粒子の位置座標の変化に基づき時間の関数として定義される．典型的には $F_s(k,t) = 1/e$ となる時間 τ_α がガラスの構造緩和過程を表すと考えられており．α-緩和時間と呼ばれる．本研究の性能測定で用いるのは $T = 0.47$ ($\tau_\alpha = 616.0$) であり，対応する線を赤色で示した．

対して，初期速度を目的温度に整合するガウス分布で生成し直したパターンを複数作成（本研究では 32 通りとする）するものである．同一配座アンサンブルを使用する背景には，ガラス形成液体における粒子運動の不均一さ（動きやすい粒子と動きにくい粒子の区別）が，粒子の配置に強く影響される一方，初期速度にはあまりつよく左右されないと考えられているからである．同一配座アンサンブルを利用することによって，構造緩和過程に伴って発生する粒子の運動性に対して，それぞれ独立なランごとに偶発的に生じた運動を取り除いた運動傾向を把握することができるのである．先述の通り 100 通りの異なる $t = 0$ の配置を用意しているので，本ケースに対しては全部で 3200 個のシミュレーションを実施する．先行研究 [7] ではデータセットは一定圧力条件の計算により作成されているが，本研究における $t = 0$ 以降の同一配座アンサンブルシミュレーションは一定体積条件であり，この違いは学習結果に大きく影響する．本研究では，$t = 0$ 以降は熱浴を切断している．熱浴の有無に構造緩和の動力学が影響されないことはガラス物理学の分野では既に確立された事実であり [17]，本研究の結果に対する熱浴の有無による影響はない．

データセットの作成に際してはそれぞれの原子の軌道を保存することになるが，4096 粒子からなる小規模系とはいえ，3200 通りの軌道を取得するだけでかなりのデータ量となる．データセットとしては，緩和時間 $10^{-3} \times \tau_\alpha$ 付近を最小，$100\tau_\alpha$ 付近を最大とする範囲の時間 t に対して対数的な尺度でチェックポイントを設け，そのチェックポイントに限定して各原子の軌道データを保存，学習に使用している．

3. 機械学習

3.1 グラフニューラルネットワークモデル

グラフニューラルネットワーク (Graph Neural Network, GNN) は，ある集合を多数の頂点 (node) とそれらの間の関係を表現する辺 (edge) として表現するグラフ上のデータに対して深層学習を行う手法である．GNN においては各頂点に結びついた辺あるいは隣接点からの情報を集約する数学的手続きを行うことによって頂点に割り当てられた特徴量が更新される．

本研究では，先行研究 [7] において用いられたのと同じ GNN を利用する．図 2 および図 3 にその概要を説明する図を掲載した．この GNN では，それぞれの頂点が原子に対応する．隣接する原子間の関係が辺として表現されるが，本研究でも先行研究 [7] でも 2 原子 i, j 間の $t = 0$ での距離が $r_{ij}(t = 0) < 2.0\sigma_{AA}$ であれば辺を結ぶ形でグラフを定義している．この GNN はエンコーダーとデコーダー (図 3 中では “EN” および “DE” と表記）を伴い，それぞれの中では ReLU を活性化関数とする (64+64) 個の多層パーセプトロン (MLP) が配置されている．隠れ層では，辺にアサインされた MLP 上で，隣接する頂点およびエンコーダーからの情報がアップデートされる．また引き続いて頂点にアサインされた MLP 上で，それを取り囲む辺に対応した MLP からの出力およびエンコーダーからの情報がアップデートされる．これを 7 回繰り返したのち，デコーダー側で特徴量が抽出され，MLP から出力される際には 1 個の量に落とすための全結合層が追加されている．

本研究および先行研究 [7] で用いた GNN におけるエンコーダーの入力情報，デコーダーからの出力情報について述べる．頂点 (node) のエンコーダーには，入力情報として 3 次元 KALJ 液体を構成する粒子の種類が A,B いずれであるかが入力されるが，これは学習に大きく影響しない付加的情報である．原子 i, j 間に割り当てられた辺 (edge) のエンコーダーからは，これら 2 原子間の 3 次元相対座標

$$\boldsymbol{r}_{ij}(0) = \boldsymbol{r}_i(t = 0) - \boldsymbol{r}_j(t = 0) \tag{3}$$

として FP32 の精度の浮動小数として入力される．

一方，デコーダーは頂点のみにアサインされている．本研究では，それぞれの頂点に対応した原子の易動度 $\Delta r_i(t)$ をスカラー量である距離

$$\overline{\Delta r_i(t)} = \langle |\boldsymbol{r}_i(t) - \boldsymbol{r}_i(0)| \rangle \tag{4}$$

として評価し，これを各頂点にアサインされた目的特徴量とする（$\langle\cdot\rangle$ は同一配座アンサンブル平均を示す）．このモデルについては，我々自身の側で現在修正を考えているところであるが，頂点 (node) 側ではなく辺 (edge) 側に対してデコーダーを配置，i, j 粒子間の距離変化に対する学習

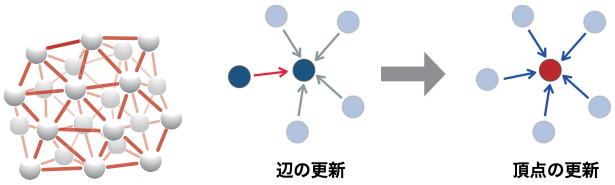

図 2 本研究で用いた GNN の基本的な構成．頂点が原子に対応し，それらの間の位置関係を辺にエンコードした GNN を利用する．GNN の学習は［頂点 → 辺］，［辺 → 頂点］でのメッセージ伝達を通じて進行する．

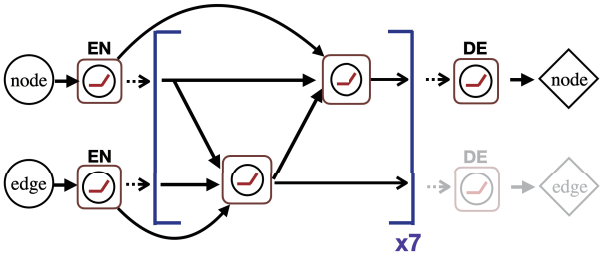

図 3 GNN のブロック図．それぞれの頂点とそれぞれの辺に，(64×64) の多層パーセプトロン (MLP) が配置される．丸四角形状で囲われた各領域は，その MLP を表す．頂点および辺にはそれぞれ粒子種と原子ペアの相対位置関係がエンコードされる．学習によるアップデートの際には，辺の更新についで頂点の更新されるステップが 7 回繰り返される．なお，本稿には記述しないが，筆者らは頂点ではなく辺にデコーダーを配置したモデルも別途作成し興味深い結果を得ている．

を行うと，物理的に異なった情報の抽出ができ，新たな予測ができることを見出している．

特徴量の更新のステップにはグラフ上の特徴量と近傍の辺（点）からの情報を伝搬するグラフ畳み込み層 [18] が利用されることが多い．本研究および先行研究 [7] では，同じよりシンプルな更新ステップを用いた GNN を利用している．この GNN は 2016 年前後に提案された Interaction Network [19] と呼ばれる GNN に近い．Interaction Network は素粒子物理学実験などで最近でも利用事例がある [20]．本研究および先行研究 [7] で利用されている GNN では，(1) 隠れ層内での再帰的な繰り返し更新を行うこと (2) MLP は頂点および辺の特徴量に対応してアサインされるが，全体を特徴づけるグローバル特徴量を使用しないこと，の 2 点がオリジナルの GNN [19] と異なっている．

3.2 GNN による学習と予測事例

我々は，この GNN を使用した予測について，先行研究 [7] の再現，およびこの予測性能を向上させる方法について物理学的な観点からの研究を合わせて展開している．本節では，先行研究 [7] の再現結果について簡単に触れておきたい．前節で紹介した GNN の学習を実施すると，そのモデルは学習した易動度 $\overline{\Delta r_i(t)}$ の予測を，独立なシミュレーションから作成された $t = 0$ でのスナップショット 1 枚から予測することが可能となる．図 4 に，上述の GNN

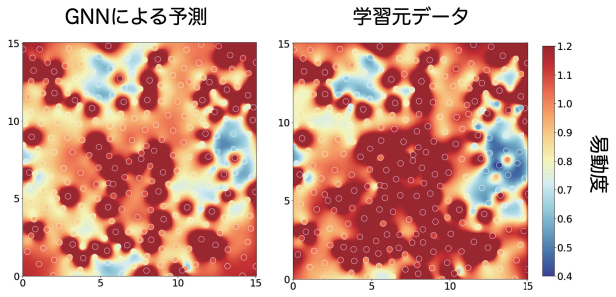

図 4 各原子の易動度 $\overline{\Delta r_i(t)}$ の $t = 0$ からの予測結果 (左) と実際にシミュレーションをした結果 (右)．学習の対象となっているのは 3 次元 KALJ 液体であるが，2 次元の断面図を示している．カラーマップは粒子の易動度を示しており，赤いところほど大きく動き，青いところほど動きがすくないことを示している．なお，この図はデモンストレーションであるため，先行研究 [7] のデータセットによる結果を示したが，我々自身のデータセットを使用しても結果は予測結果は同様である．

を使用して頂点上での易動度 $\overline{\Delta r_i(t)}$ を予測した結果を，実際に同一配座アンサンブルのシミュレーションをすることで得られる易動度 $\overline{\Delta r_i(t)}$ と並べてカラーマップとした結果を示している．実際には 3 次元に対する計算が行われているが，2 次元状の断面を切り出す形でデータを表示している．（なお，2 節に述べた我々自身のデータセットではなく，先行研究 [7] の著者が提供したものを使用している）．左右の比較から，空間的に不均一な易動度の分布の傾向が非常によく再現されていることがご理解いただけると思う．この予測がどの程度正確であるのかについては，ピアソン相関係数などを用いて評価が可能であり，実際にこの GNN による予測の性能は，これまでに提案されたことのある手法と比較して定量的にベストに位置することが先行研究 [7] で明らかにされている．

我々は，この GNN モデルをさらに拡張し，GNN の辺に対して定義された特徴量を導入するなどの試みを行っている．その結果，緩和時間 τ_α より短い時間においては，GNN による予測性能をさらに向上させることが可能となることが見出された．この内容については別途の出版予定があるため本稿では触れないが，研究会当日に時間の許す範囲で紹介する予定である．.

3.3 PyTorch を利用した実装

先行研究 [7] を実施した DeepMind と Google Brain の研究グループは，研究に使用した TensorFlow 向けの GNN 学習コードを公開している[*1]．しかし，一部に TensorFlow 2 系以上では動作しない部分があるため，最新の GPU での実行は難しくなっている．同じ著者らは 2021 年 6 月に同じリポジトリ JAX で動作するルーチンを追加しており，我々はこのルーチンを利用して研究を行った．JAX [21]

[*1] https://github.com/deepmind/deepmind-research/tree/master/glassy_dynamics

は，自動微分 (autograd)・高速線形代数 (XLA) の計算を GPU, TPU などで高速に実行するためのフレームワークである．高速化をかけたいコードの部分に対して指示文あるいはラッピング関数などを挿入することによって，Just in time (jit) コンパイルや並列化を施した上での演算をさせることができる．JAX はあくまでも自動微分等の高速計算のためのライブラリであり，これを用いてデータを学習するためには別のライブラリの導入が必要であり，ニューラルネットワーク構築ライブラリ dm-haiku [22] や GNN 構成ライブラリ Jraph [23]，勾配伝搬および最適化用ライブラリの Optax [24] とともに使用する．

この JAX コードは我々の現在の研究の用途に足りるものとなっているが，我々は今後の研究展開をを見据えて PyTorch 上での実装を進めた．GNN による学習を PyTorch 上で実施するためのツールとして最初に普及したのは Deep Graph Library (DGL)[*2] [25] であり GNN に頻出する疎行列演算速度に優位性があると言われている．一方，近年は PyTorch Geometric (PyG) [26] が急速に普及を見せており，原稿執筆時点（2022 年 2 月）では DGL よりも優位なシェアを占めるに至っている．我々は両者を実装して検討する中で後者を用いた実装が完遂できたことにより，今回その性能測定を行っている．

JAX コードとの比較を考えたとき，PyG を利用する利点は次が挙げられると考えている．

- 先行研究 [7] グループの提供しているコードではデータのミニバッチ化が行われていない．グラフデータのミニバッチ化においては，データごとに異なる数の辺（加えて，本研究に限らない場合には頂点の数も異なる）を有したデータの結合が必要となる．PyG は，この配列の結合をフレームワークとして実現し，メモリ管理の上でも効率的な実装を提供する．またこのようにミニバッチ化されたグラフデータを，PyTorch の API として備わった DataLoader 上で，あたかも PyTorch 上で操作しているように使用することを可能にする．
- PyTorch においてはデータ並列実行のための API が備わっている（`DataParallel` および `DistributedDataParallel`）．前項の DataLoader を使用すれば，PyG でもそのままデータ並列が実行できる．
- PyTorch バージョン 1.8 以降では ROCm 上での動作が可能になっており，AMD GPU 上で容易に走らせることができるようになっている．JAX も様々なアーキテクチャーの GPU で動作することが謳われており我々もリビルドして動作さあせることを試みたが，現時点では AMD MI100 での学習には成功していない
- GNN にはいくつかの拡張が考慮されているが，この中でもグラフ構造の動的な変化を取り入れられる Dynamic Graph（例として文献 [27]）や，自己注意機構を Transfomer や取り入れた GNN [28] への拡張は重要な課題と我々は考えている．これらの拡張を行うには，豊富なコードスタックを有する PyG の方が有利であると考えられる．

JAX で用いられている GNN 構成ライブラリ jraph においては，Interaction Network の作成に必要な API が備わっている．一方，我々の現時点のやや不確かな理解の範囲では，辺と頂点の間のアップデートを互いに行う形式でのメッセージ伝搬用に直接使用できる API が PyG には存在していない．我々は UC San Diego の講義ノート [29] に付属してウェブ公開されている Jupyter Notebook コードを参照しこれを大きく変更していく形で，PyG への Interaction Network の作成を実装した．コードの詳細の説明については割愛するが，後日コードの公開を予定している．

4. 性能測定

本節では，GNN による分子動力学予測に対するデータ学習を行う際の性能測定結果を報告する．

今回の測定対象としたのは学習タスクを実行している時間のみであり，予測（推論）については計測の対象としていない．またストレージからホスト CPU にデータを転送する時間を測定時間からは除外してある．学習対象のデータは，100 通りの初期条件スナップショットに対する同一配座アンサンブルから得られた各粒子の易動度であり，$t = 0$ のスナップショットから，系の緩和時間（τ_α）先に相当する時間 $t = 616.0$ における各粒子の易動度を予測するための GNN の訓練タスクを実施した．学習は 1000 エポックにわたって実施するが，今回の学習に対しては学習曲線は終了前に十分収束をする．

JAX での学習時ではデータのミニバッチ化は行っておらず，PyTorch の学習時には 5 個（4.2 節のデータ並列については 2 個）のスナップショットに対するグラフネットワークをまとめてミニバッチ化している．なお，主として環境構築および使用ライブラリ間のバージョン整合性の問題があるため，ホストマシンを揃えた学習性能測定をすることは現時点ではできていない．表 1 には実行環境を一覧として示す．

4.1 単体性能

まず，単体 GPU における学習性能の比較を，先行研究 [7] の著者らによるコードと今回開発した PyG コードとの間で比較する．

1000 エポックの学習を実施にかかった時間を測定したところ，１エポックあたりの学習時間は次の通りであった．

[*2] `https://github.com/dmlc/dgl/`

表 1 性能測定環境の一覧．# GPUs はノードあたりに接続された GPU 台数を示す．AMD MI100 ではノードあたり 4 基搭載となっているが，今回は 1 基での測定のみを行った．

節	GPU	#GPUs	ホスト CPU	プラットフォーム
4.1,4.3 ローカルクラスタ	NVIDIA A100 40GB PCIe	1	AMD EPYC 7443 1 CPU, 24 コア	CUDA 11.3, PyTorch 1.10
	AMD Instinct MI100	4	AMD EPYC 7662, 2 CPUs, 128 コア	ROCm 4.5.2, PyTorch 1.10
4.2 Wisteria-A	NVIDIA A100 40GB SXM	8	Intel Xeon Platinum 8360Y 2 CPUs, 72 コア	CUDA 11.1, PyTorch 1.8

- JAX – `2.350 sec`
- PyG – `2.024 sec`

両者の間には 15%程度の性能差が存在している．この原因としては，PyG の方が内部でグラフのデータをミニバッチ化し，配列にもあまりがでない形で無駄がない形で処理しているからと推測される．JAX で使用している GNN ツール jraph がグラフを作成する際のデータ構造は，edge の情報を格納する配列長を固定（本研究の差には 180,000 に設定した）しており，余った部分はダミーのノードを参照する．このために一定数の余分な演算が発生していると思われる．

4.2 データ並列性能 (Wisteria-A)

PyG を用いて実装された GNN においては，DataLoader を使用してミニバッチ化されたデータに対して PyTorch と同等のレベルでのデータ並列学習を実行することができる．本項目ではプロセスベースのデータ並列手法である Distributed Data Parallel を実装してマルチ GPU による学習性能のストロングスケーリング測定を行った．この性能測定は東京大学情報基盤センターに 2021 年 5 月に導入され現在運用稼働中の Wisteria/BDEC-01 システムの，NVIDIA A100 Tensocore GPUs (40GB, SXM) をノードあたり 8 基ずつ搭載した Aquarius サブシステムを用いて実施した．ノード間は Infiniband HDR で接続されている．

PyTorch における `torch.distributed` を用いたデータ並列の実行に当たっては，通信バックエンドを MPI とするか NCCL とするかの選択を迫られる．自身で PyTorch のソースビルドが実行できる場合には，前者の方が環境設定は容易な場合がある．ただし，Wisteria-A においては，現在通信バックエンドとして NCCL 2.8.4 が HDR Infiniband ノード間通信を利用できるように動作する状態となっており，今回こちらを利用することとした．Wisteria/BDEC-01 では富士通社製の TCS スケジューラーが採用されており，ジョブスクリプトはそれに合わせた形での記述が必要である．このジョブスクリプト記載方法について付録の A.1 節に示した．この方法は，PyG に限らず PyTorch の Distributed Data Paralell に共通のものであるが，PyTorch のバージョンアップに伴い方法が変更になる可能性がある．

GPU の台数を変化させていったときの測定結果は図 5 に示している．1 GPU での PyG (PyTorch) の実行も，Distributed Data Parallel 用に書き換えたコードで実施している．4.1 節におけるのに比較して，15% 程度，学習性能が低下した．これは（データ並列のために）ミニバッチのバッチサイズを下げて測定を行ったことと，`DistributedSampler` を使用したことによるオーバーヘッドの発生が原因として想定される．図中には 1GPU で JAX により学習を行ったときの学習時間を参考情報として共にプロットしているが，こちらでも 4.1 節におけるのに比して若干の性能低下があった（共用ノードで測定したための可能性がある）．一方，マルチ GPU での学習性能についてのストロングスケーリングは，良好であることが見て取れる．GPU 64 基を使用したときの並列化効率は約 72%であり，4.1 節のシングル GPU 向けコードの実行性能と比較したときの並列化効率は約 63%でとなっている．もともと，取り扱っている GNN においては辺の数が 170000 以上であり，かなり巨大なネットワークになっており，並列性能を向上させやすい状態となっている．現時点ではデータ数は少し絞った状態で計測しているが，データセットのサイズを大きくした場合には並列化効率はさらに良好になると思われる．

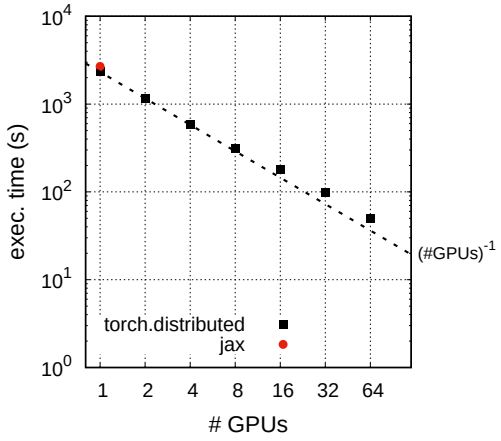

図 5 PyG による分散データ並列処理性能のプロット．系列名の `torch.distributed` は列が PyG (PyTorch) の Distributed Data Parallel を利用した測定結果である．横軸は GPU 数であり，縦軸は 1000 エポック学習に要する時間を秒の単位で記載している．1GPU のところに，同一の環境で計測した JAX による学習時間を赤丸で共にプロットしている．

4.3 AMD MI100 GPU での学習性能

科学技術計算向けおよび機械学習・深層学習向けのワーク

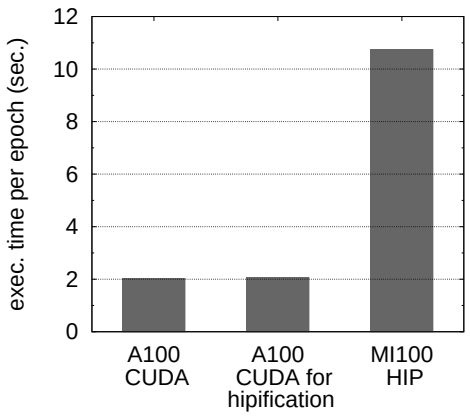

図 6 GPU 1基によるエポックあたりの学習の所要時間．左側は NVIDIA A100 での測定結果，中央は HIP 化用に修正した CUDA コードによる NVIDIA A100 での測定結果，右が HIP 化されたコードによる AMD Instinct MI100 での測定結果である．

ロード双方を処理できるプラットフォームとして NVIDIA 社の GPU は先導的な位置にあったが，科学技術計算の用途では他社による GPU も追随している．筆者らの所属する東京大学情報基盤センターでも性能評価が進められている [30]．今回，最新の GPU として AMD MI100 を用いて GNN の学習の性能測定を行ったので，NVIDIA A100 GPU との結果の比較を本節に示す．なお，我々が知る限りでは，GNN を AMD MI100 GPU を用いて学習した事例の他の報告は未だ存在しない．

PyTorch ではバージョン 1.8 以降，AMD MIOpen および RCCL ライブラリ上で動作する大規模学習向け AMD ROCm Platform 向けパッケージが提供されている．これを用いることで，AMD MI100 GPU での学習が可能になる．PyG を使用するに際しては，PyG から参照されるツールである PyTorch Sparse（疎行列演算支援）および PyTorch Scatter（スパース更新演算支援）について ROCm 対応のインストールを行うには，それぞれのセットアップスクリプトを自ら編集する必要がある．これに必要な環境構築手順については，付録の A.2 節に詳述する．この構築手順の中では PyTorch Sparse および PyTorch Scatter 中にある CUDA コードをセットアップスクリプトから自動 HIP 化する手順があり，一部は `hipcc` によるコンパイルが通らない．このコンパイルが通らない CUDA コードを書き換えて HIP 化可能とする必要がある．この書き換えの影響についても検証するため，HIP 化可能な CUDA コードからセットアップされた PyG の学習性能についても検証した．

図 6 には，棒グラフとしてエポックあたりの学習に要する時間をプロットしている．PyG 関連ツールを HIP 化可能なように書き換えを行った場合でも，性能差 1% 程度に収まっており，大きな影響は出ていない．一方，AMD MI100 での実行時間は，これらと比較して約 5.2 倍遅いという結果になった．推測するに，NVIDIA A100 における学習では TensorFloat32 (TF32) 不動小数点の行列演算に対して Tensor コアが大きく活用されていることが原因となっているかもしれない．NVIDIA A100 における TF32 の理論ピーク性能が 156 TFlops であるの対して，Tensor コアを有しない AMD MI100 では FP32 での行列演算性能ピークが 46.1 TFlops にとどまる．ただし，この性能差について議論するためには，PyTorch 実行時のプロファイリング情報に踏み込んだ解析が必要であり，今後調査を継続する予定である．

5. 結論

近年利用が広がったグラフニューラルネットワークは，頂点と辺という構造をそのまま粒子の配置と隣接関係に置き換えることができることから，分子シミュレーションの情報を埋め込むことができ，各種の利用が広がっている．本稿では，極めて遅い緩和を示すために何十万ステップという多数の時間積分を伴うガラスの分子動力学計算の結果を，1時点の粒子配置の情報のみからグラフニューラルネットワークを用いてよく予測できることを，先行研究 [7] に基づく形で紹介した．そして今後のさらなる研究展開をに向けて，グラフニューラルネットワークを TensorFlow/JAX から PyTorch + PyTorch Geometric に移植を行った．これにより，高効率なデータ並列計算や AMD 社製の GPU でのポーティングが可能となったため，それらについての性能評価結果を報告した．今回開発した PyTorch コードおよびデータセットについては，別途準備中のプレプリントとと同時に公開する予定である．

謝辞 グラフニューラルネットワークに関わる議論・助言をしていただいている鈴村豊太郎（東大院情報理工・東大情基セ）・華井雅俊（東大情基セ）の両氏に心より感謝申し上げる．また，GPU 利用に関して三木洋平氏（東大情基セ）および山崎和博氏（NVIDIA 合同会社）からの助言に感謝する．本研究は科学研究費補助金（課題番号 19H05662）に主な支援を受けた．本研究の一部について，筆頭著者は科学研究費補助金（課題番号 18H01188）の支援を受けている．本研究の一部は，学際大規模情報基盤共同利用・共同研究拠点，および，革新的ハイパフォーマンス・コンピュー ティング・インフラ（JHPCN-HPCI 課題番号：jh210017-MDH および jh210051-MDH）の支援を受けた．また，計算物質科学スーパーコンピュータ共用事業より東京大学物性研究所スーパーコンピューター（システム B,C）および東北大学金属材料研究所 MASAMUNE-IMR の計算資源の提供を受けたことを感謝する．

参考文献

[1] Wang, H., Zhang, L. and Han, J.: DeePMD-kit: A deep learning package for many-body potential energy representation and molecular dynamics, *Computer Physics Communications*, Vol. 228, pp. 178–184 (online), available from ⟨http://dx.doi.org/10.17632/hvfh9yvncf.1⟩ (2018).

[2] Takamoto, S., Shinagawa, C., Motoki, D., Nakago, K., Li, W., Kurata, I., Watanabe, T., Yayama, Y., Iriguchi, H., Asano, Y., Onodera, T., Ishii, T., Kudo, T., Ono, H., Sawada, R., Ishitani, R., Ong, M., Yamaguchi, T., Kataoka, T., Hayashi, A. and Ibuka, T.: PFP: Universal Neural Network Potential for Material Discovery, *arXiv preprint arXiv:2106.14583* (2021).

[3] Csányi, G., Bernstein, N. and Kermode, J. R.: libAtoms/QUIP (online), available from ⟨`http://github.com/libAtoms/QUIP`⟩.

[4] Jia, W., Wang, H., Chen, M., Lu, D., Lin, L., Car, R., Weinan, E. and Zhang, L.: SC '20: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–14 (2020).

[5] Invernizzi, M., Piaggi, P. M. and Parrinello, M.: Unified Approach to Enhanced Sampling, *Physical Review X*, Vol. 10, pp. 041034/1–18 (2020).

[6] Bolhuis, P. G. and Swenson, D. W.: Transition Path Sampling as Markov Chain Monte Carlo of Trajectories: Recent Algorithms, Software, Applications, and Future Outlook, *Advanced Theory and Simulations*, Vol. 4, pp. 2000237/1–14 (2021).

[7] Bapst, V., Keck, T., Grabska-Barwińska, A., Donner, C., Cubuk, E. D., Schoenholz, S. S., Obika, A., Nelson, A. W., Back, T., Hassabis, D. and Kohli, P.: Unveiling the predictive power of static structure in glassy systems, *Nature Physics*, Vol. 16, pp. 448–454 (2020).

[8] Widmer-Cooper, A., Perry, H., Harrowell, P. and Reichman, D. R.: Irreversible reorganization in a supercooled liquid originates from localized soft modes, *Nature Physics*, Vol. 4, pp. 711–715 (2008).

[9] Cubuk, E. D., Schoenholz, S. S., Rieser, J. M., Malone, B. D., Rottler, J., Durian, D. J., Kaxiras, E. and Liu, A. J.: Identifying structural flow defects in disordered solids using machine-learning methods, *Physical Review Letters*, Vol. 114, pp. 108001/1–5 (2015).

[10] Schoenholz, S. S., Cubuk, E. D., Sussman, D. M., Kaxiras, E. and Liu, A. J.: A structural approach to relaxation in glassy liquids, *Nature Physics*, Vol. 12, pp. 469–471 (2016).

[11] Kob, W. and Andersen, H. C.: Testing mode-coupling theory for a supercooled binary Lennard-Jones mixture: The van Hove correlation function, *Physical Review E*, Vol. 51, pp. 4626–4641 (1995).

[12] Toxvaerd, S. and Dyre, J. C.: Communication: Shifted forces in molecular dynamics, *The Journal of Chemical Physics*, Vol. 134, pp. 081102/1–4 (2011).

[13] Shimada, M., Mizuno, H. and Ikeda, A.: Anomalous vibrational properties in the continuum limit of glasses, *Physical Review E*, Vol. 97, pp. 022609/1–9 (2018).

[14] Nosé, S.: A unified formulation of the constant temperature molecular dynamics methods, *The Journal of Chemical Physics*, Vol. 81, pp. 511–519 (1984).

[15] Binder, K. and Kob, W.: *Glassy Materials and Disordered Solids*, World Scientific (2011).

[16] Widmer-Cooper, A., Harrowell, P. and Fynewever, H.: How reproducible are dynamic heterogeneities in a supercooled liquid?, *Physical Review Letters*, Vol. 93, pp. 135701/1–4 (2004).

[17] Gleim, T., Kob, W. and Binder, K.: How Does the Relaxation of a Supercooled Liquid Depend on Its Microscopic Dynamics?, *Physical Review Letters*, Vol. 81, pp. 4404–4407 (1998).

[18] Kipf, T. N. and Welling, M.: Semi-Supervised Classification with Graph Convolutional Networks, *arXiv:1609.02907*, (online), available from ⟨http://arxiv.org/abs/1609.02907⟩ (2016).

[19] Battaglia, P. W., Pascanu, R., Lai, M., Rezende, D. and Kavukcuoglu, K.: Interaction Networks for Learning about Objects, Relations and Physics, *Adv. Neural Inf. Process. Syst.*, Vol. 29, pp. 4502–4510 (online), available from ⟨http://arxiv.org/abs/1612.00222⟩ (2016).

[20] DeZoort, G., Thais, S., Duarte, J., Razavimaleki, V., Atkinson, M., Ojalvo, I., Neubauer, M. and Elmer, P.: Charged Particle Tracking via Edge-Classifying Interaction Networks, *Computing and Software for Big Science*, Vol. 5, pp. 26/1–13 (2021).

[21] Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S. and Zhang, Q.: JAX: composable transformations of Python+NumPy programs (online), available from ⟨`http://github.com/google/jax`⟩.

[22] Hennigan, T., Cai, T., Norman, T. and Babuschkin, I.: Haiku: Sonnet for JAX (online), available from ⟨`http://github.com/deepmind/dm-haiku`⟩.

[23] Godwin, J., Keck, T., Battaglia, P., Bapst, V., Kipf, T., Li, Y., Stachenfeld, K., Veličković, P. and Sanchez-Gonzalez, A.: Jraph: A library for graph neural networks in jax. (online), available from ⟨`http://github.com/deepmind/jraph`⟩.

[24] Hessel, M., Budden, D., Viola, F., Rosca, M., Sezener, E. and Hennigan, T.: Optax: composable gradient transformation and optimisation, in JAX! (online), available from ⟨`http://github.com/deepmind/optax`⟩.

[25] Wang, M., Zheng, D., Ye, Z., Gan, Q., Li, M., Song, X., Zhou, J., Ma, C., Yu, L., Gai, Y., Xiao, T., He, T., Karypis, G., Li, J. and Zhang, Z.: Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks, *arXiv preprint arXiv:1909.01315* (2019).

[26] Fey, M. and Lenssen, J. E.: Fast Graph Representation Learning with PyTorch Geometric, *ICLR Workshop on Representation Learning on Graphs and Manifolds* (2019).

[27] Pareja, A., Domeniconi, G., Chen, J., Ma, T., Suzumura, T., Kanezashi, H., Kaler, T., Schardl, T. B. and Leiserson, C. E.: EvolveGCN: Evolving Graph Convolutional Networks for Dynamic Graphs, *Proceedings of the Thirty-Fourth AAAI Conference on Artificial Intelligence* (2020).

[28] Hu, Z., Dong, Y., Wang, K. and Sun, Y.: Heterogeneous Graph Transformer, *WWW '20: The Web Conference 2020, Taipei, Taiwan, April 20-24, 2020*, ACM / IW3C2, pp. 2704–2710 (online), available from ⟨https://doi.org/10.1145/3366423.3380027⟩ (2020).

[29] Duarte, J. and Würthwein, F.: Particle Physics and Machine Learning (online), available from ⟨`http://jduarte.physics.ucsd.edu/`⟩.

[30] 三木洋平, 塙敏博: AMD MI100 に向けた N 体計算コードの移植と性能評価, 研究報告ハイパフォーマンスコンピュー ティング (HPC), Vol. 2021-HPC-180, pp. 1–10 (2021).

付　　録

A.1 Wisteria/BDEC-01 (Aquraius) における PyTorch 分散データ並列実行方法

PyTorch において NCCL バックエンドで分散データ並列通信するプロセスを立ち上げる方法はいくつかあるが，本稿の Wisteria-A における性能評価で使用している PyTorch 1.8 スーパーコンピューターシステムで一般的なジョブスケジューラーとの親和性が比較的良い手法は，各ノードから `torch.distributed.launch` コマンドを実行するという方法である．

以下にはジョブスクリプトから実行部分のみを書いたものを記載する．実際の実行に際しては Environment module などを利用することにより CUDA, CUDNN, NCCL などの適切な環境変数を設定することが必要である．

```
#!/bin/bash
for IP_ADDR in "([IPaddresses])"
do
    pjrsh ${IP_ADDR} python -m torch.distributed.launch --nproc_per_node=8 --nnodes
       =8 --node_rank=${count} --master_addr="${MASTER_ADDR}" --master_port=21135
       run.py &
    count=‘expr $count + 1‘
done
wait;
```

上においては `MASTER_ADDR` はランク 0 のプロセスを有する計算ノードの IP アドレス，`IP_ADDR` は並列対象となっている計算ノードの IP アドレスであり，富士通社が提供しているジョブスクリプト関連コマンドにより取得したものである．`pjrsh` は TCS スケジューラー上でのマスターノードから他の計算ノードに対するリモートシェルコマンドである．`--master_port` には利用可能なポート番号を適切に設定する．

なお，現在 PyTorch では分散プロセスの立ち上げるコマンドとして `torchrun` コマンドへの移行が進められており，`torch.distributed.launch` は今後非推奨となる予定である．`https://pytorch.org/docs/stable/distributed.html` の記載を参照のこと．

A.2 AMD MI100 向け PyTorch Geometric の環境構築

本節では，今回使用した GNN 学習ライブラリ PyTorch Geometric を AMD MI100 で動作させるために行った環境構築手順を箇条書きの形で以下に記述する．

- `pip` 仮想環境を作成し，PyTorch 公式ホームページ[*3]の指示に従って PyTorch をインストールする．本研究では `Python 3.8.12` 上で ROCm 4.2 対応の PyTorch 1.10.2 を導入したが，実際の HIP コンパイラは上位互換であるバージョン 4.5.2 を使用した．この PyTorch を `pip` インストールする場合，依存関係を持つ `typing-extensions`，`numpy`，`torchvision` などが共にインストールされる．
- PyG は追加ライブラリとして PyTorch Sparse（疎行列演算支援）および PyTorch Scatter（スパース更新演算支援）を利用する．これらは通常 `pip` コマンドによるインストールが可能であるものの，ROCm 上で動作させる場合には，C++/CUDA で書かれた CUDAExtension ソースコードを HIP に変換する必要があるため，今回はソースをダウンロードした上でビルドを実施する．
  (1) PyTorch Sparse を直接 GitHub よりダウンロードする．
  ```
  > git clone https://github.com/https://github.com/rusty1s/pytorch_sparse.git
  ```
  PyTorch Sparse などの PyTorch 関連ライブラリでは CUDAExtension ソースコードは `csrc` という名称のディレクトリ内に格納されている．
  (2) インストールスクリプトにある `setup.py` のうち，`CUDA_HOME` などいくつかの環境変数を ROCm に対応した変数 `ROCM_HOME` に書き換えることで，ROCm 環境が利用できるようにする．書き換え箇所を以下に示す．

[*3] `https://pytorch.org/get-started/locally/`

```
    ’’’冒頭省略’’’
    import torch
    from torch.__config__ import parallel_info
    from torch.utils.cpp_extension import BuildExtension
    from torch.utils.cpp_extension import CppExtension, CUDAExtension,
        CUDA_HOME

    # helper code - see wiki on GitHub ROCmSoftwarePlatformom/PyTorch
    is_rocm_pytorch = False
    if torch.__version__ >= ’1.5’:
        from torch.utils.cpp_extension import ROCM_HOME
        is_rocm_pytorch = True if ((torch.version.hip is not None) and (
            ROCM_HOME is not None)) else False

    WITH_ROCM = torch.cuda.is_available() and ROCM_HOME is not None
    suffices = [’cpu’, ’cuda’] if WITH_ROCM else [’cpu’]
    if os.getenv(’FORCE_CUDA’, ’0’) == ’1’:
        suffices = [’cuda’, ’cpu’]
    ’’’以下省略’’’
```

(3) 同じスクリプト中で CUDA オプションが変数 nvcc_flags として指定されている部分を置き換える．また，MI100 向けの命令を生成するために該当するマシンタイプ gfx908 を指定する．

```
[setup.py:L77]
- nvcc_flags += [’--expt-relaxed-constexpr’, ’-O2’]
+ nvcc_flags += [’-O2’, ’--amdgpu-target=gfx908’]
[setup.py:L86]
- if sys.platform == ’win32’:
-     extra_link_args += [’cusparse.lib’]
- else:
-     extra_link_args ] += [’-lcusparse’, ’-l’, ’cusparse’]
+ if sys.platform == ’win32’:
+     extra_link_args += [’hipsparse.lib’]
+ else:
+     extra_link_args += [’-lhipsparse’, ’-l’, ’hipsparse’]
```

(4) セットアップスクリプトを以下のコマンドにように実行するとビルドが開始される．もともと csrc/cuda ディレクトリに配置されていた CUDA カーネルが CUDAExtension によって HIP カーネルへと自動変換されたコードが生成，csrc/hip ディレクトリに格納されたのちにコンパイルされる．

```
> python3 setup.py build
```

こうして書き換えられた HIP コードのいくつかはビルドが通らない．ビルドする過程で見つかった全ての箇所について，コンパイルが通るようにソースコードを書き換えることで，全てのソースコードのコンパイルを通していく．PyTorch Sparse については次の書き換えを行った．

- いくつかの CUDA カーネルで書かれた L1 キャッシュ指示の組み込み関数 __ldg(const *ptr) が HIP 化されない．この関数に対するラッパーとして，ソース中で単に自身のポインタ（*ptr）を返すテンプレート関数を定義しコンパイル阻害要因を回避する．
- CUDA のバージョンを返り値とする関数を記述した version.cpp のソースコードおいて CUDA_VERSION を HIP_VERSION に置換する．

(5) コンパイルが終わったら，次のインストールコマンドを実行する．

```
> python3 setup.py install
```

(6) PyTorch Scatter[*4]についても PyTorch Sparse に対して上記のように行ったのと全て同様の手順でビルドおよびインストールを実施する．

[*4] https://github.com/rusty1s/pytorch_scatter

- 以上が完了したら，PyG を pip インストールする．依存関係のあるライブラリが場合によっては多数同時インストールされる．

```
> pip3 install torch-geometric
```

- 上述の手順により今回インストールされた PyTorch および PyG 関連ツールのバージョンはそれぞれ次の通りである．

```
torch             1.10.2+rocm4.2
torch-geometric   2.0.3
torch-sparse      0.6.12
torch-scatter     2.0.9
```

（以上）