Vol.2017-HPC-161 No.13
2017/9/20

NVLink と Unified Memory を利用した OpenACC と OpenMP4.x による GPU プログラミング

土井　淳[†1]

概要：近年，GPU のような加速装置を利用したスーパーコンピューターが主流となりつつある．しかしながら，加速装置を利用するためのプログラミングはやや複雑であり，十分な性能を得るために，ソースコードの大幅な書き換えが必要となることがある．このような加速装置を簡単に扱えるような手法として OpenACC や OpenMP のような指示文を用いて処理を加速装置にオフロードする方法によってソースコードの書き換えを最小限に抑えた生産性の高いプログラミング環境が整いつつある．しかしながら，そのような指示文を用いる方法においても，CPU と加速装置の間のデータのやり取りを記述する必要があり，まだプログラミングの複雑さが残ってしまう．そこで，CUDA に実装されている Unified Memory の仕組みを用いることで，同一の配列を CPU と GPU でアクセスすることができ，これによってプログラミングが簡素化されることが期待されるが，明示的にデータ転送を行う場合に比べて性能が出にくい問題がある．しかし，NVLink の登場により，CPU-GPU 間のデータ転送速度が向上したことで，我々は Unified Memory に OpenACC を組み合わせることで実用的に十分な性能を得ることができることを前報告で示した．本報告では，同様に OpenMP4.x の指示文を用いていくつかの HPC アプリケーションについて GPU 対応を行い性能を調査した．OpenACC を用いた場合との性能比較とプログラミングの違いについて報告する．

キーワード：NVLink, OpenACC, OpenMP, GPU, CUDA, Unified Memory, OpenPOWER

1. はじめに

GPU に代表されるような加速装置を従来の計算機に組み合わせたハイブリッドな計算機は，電力効率の良さなどから，近年のスーパーコンピューターの主流となりつつある．このような加速装置を利用した計算機において計算を行うには，計算を行う場所のメモリにデータを置く必要があり，このためには計算機ノードの CPU と加速装置の間でデータの転送を行う必要がある．このことが，加速装置を利用したプログラミングの複雑さを増す要因の一つとなっている．また，このときの加速装置との間のデータの転送速度が，加速装置による性能向上のためのボトルネックになることもしばしばある．

加速装置を利用した計算機では，加速装置による十分な性能向上が得られることはもちろん，ユーザーにとってはプログラミングのしやすさも要求される．従来からの共有メモリ並列計算機では，OpenMP のような指示文ベースのプログラミング手法を用いることで，既存のコードを比較的簡単に並列化でき，望んだ性能を得ることができる．これは，共有メモリの仕組みによって，単一のメモリ空間でプログラミングができることの恩恵も大きい．GPU 等の加速装置を用いた計算機システムでは，OpenMP と同じように，指示文形式である OpenACC[1]や，OpenMP の加速装置向けの拡張である OpenMP4.x[2]を用いることで，比較的簡単に加速装置を利用することができる．しかしながら，データの管理はプログラマーが行う必要があり，どのデータをどのタイミングで CPU と加速装置の間で転送するかを指示文で書かなければならない．特に，複雑な構造のプログラムを移植する場合にプログラムの生産性を悪化させる原因となりかねない．

共有メモリ並列の考え方と同じように，CPU と GPU で単一のメモリ空間を扱えるようにした実装が，Unified Memory[3]である．Unified Memory を利用することで，CPU-GPU 間のデータの転送は，必要に応じて自動的に行われるようになるため，プログラマーがデータの転送や配列の確保を管理する必要がなくなるためプログラミングの生産性が改善できると考えられる．しかしながら，Unified Memory も，OpenACC や OpenMP4.x のような指示文ベースのプログラミング手法も，CUDA を用いて CPU-GPU 間のデータ転送と計算の処理を極限までに最適化したコードに比べると，CPU-GPU 間のデータ転送がネックになることが多い．

高速なインターコネクト技術である NVLink[4]の登場によって，CPU-GPU 間のデータ転送によるボトルネックを解消できることが期待される．前報告[5]では，高速なインターコネクトである NVLink を利用して，Unified Memory および OpenACC を組み合わせたプログラミングによるアプリケーションの GPU への移植手法を提案し，より少ないソースコード書き換えにより，十分な性能を得ることを実アプリケーションを用いた性能評価によって示した．本報告では，それに加えて，OpenMP 4.x 指示文を用いた GPU プログラミングと Unified Memory の組み合わせについても評価し，OpenACC との比較を行う．

二章では，OpenMP 4.x の概要と，OpenACC との違いについて説明し，Unified Memory を利用したプログラミング手法について述べる．三章では，HPCG ベンチマークと CoMD の 2 つのアプリケーションを OpenMP 4.x と OpenACC を用いて GPU 対応を行い，その性能評価について説明する．

†1 日本アイ・ビー・エム株式会社 東京基礎研究所
IBM Research – Tokyo

2. OpenMP 4.x による GPU プログラミング

2.1 OpenMP 4.x 概要

OpenMP 4.x は，従来の共有メモリ並列化（SMP）向けの OpenMP による指示文ベースのプログラミングモデルを拡張し，加速装置へのワークロードのオフローディングに対応した新たな指示文などを追加した規格であり，最新の仕様はバージョン 4.5 である．OpenACC と同じように，計算カーネルをオフロードするための指示文や，データを管理し，配列の確保や転送を行うための指示文などから構成される. OpenACC に比べると比較的最近登場した仕様であり，実装がまだ少なく，GPU 向けとして利用が可能なのは，LLVM ベースのオープンソースコンパイラである Clang コンパイラー（C/C++用）[6][7], IBM XL C/C++ Compiler version 13.1.5 あるいは，IBM XL Fortran Compiler version 15.1.5 等などがある．

2.2 OpenACC と OpenMP 4.x の実装の違い

特に大きな実装の違いは, CPU-GPU 間でのデータの管理の仕方である．OpenMP 4.x では，データはデバイスにマップして初めて使用可能となる．CPU から GPU に配列をマップすると, GPU にデータが転送され, その配列が GPU で利用可能となる．逆に，GPU から CPU へマップ（アンマップ）すると，CPU へデータが転送される．これらは内部でカウンターで管理されており，CPU から GPU へマップされるとカウンタが加算され，逆にアンマップされると減算され，カウンターが正の値のときに GPU から使用ができ，ゼロになると GPU 上のメモリが開放される．よって，プログラマーが適切なタイミングでデータをマップしないと正しく動かない．これに対して，OpenACC では，データ転送指示文を書かなくても，コンパイラーが自動的にデータ転送を行うコードを生成し，正しく動くコードが出来上がる．しかしながら，既に GPU 上にデータがある場合，OpenACC では，DATA PRESENT 指示文によって，無駄なデータ転送を抑制する必要があるのに比べて，OpenMP 4.x ではこれが不要となる．

2.3 OpenMP 4.x で Unified Memory を使用する

前述の通り，OpenMP 4.x で GPU で使用するデータはマップされている必要がある．これは Unified Memory を使用する場合でも例外ではなく，マップしないと実行時にエラーとなる．ところが，次のような内積計算を行う Fortran コードのように，MAP 指示文で，Unified Memory で確保した配列をマップしてしまうと，これらの配列のコピーが GPU のメモリ上に生成され，既に GPU 上にある元の配列からデータのコピーが発生してしまう．無駄な処理が追加されるのと，GPU 上にコピーが生成されることでメモリ使用量が増える問題がある．

```
        real(8),dimension(:),allocatable,managed::a,b
        allocate(a(n))
        allocate(b(n))
        dot = 0.d0
#ifdef _OPENACC
!$ACC DATA DEVICEPTR(A,B)
!$ACC KERNELS
!$ACC LOOP INDEPENDENT REDUCTION(+:dot)
#else
!$OMP TARGET MAP(dot) MAP(to:,a,b)
!$OMP TEAMS DISTRIBUTE PARALLEL FOR
REDUCTION(+:dot)
#endif
        do i=1,n
          dot = dot + a(i)*b(i)
        enddo
```

このソースコードでは, OpenACC の場合の例も示したが，OpenACC では，DEVICEPTR 指示文（もしくは PRESENT 指示文）を用いることで，データ転送が不要であり，そのままの配列が使用可能であることを指示できる．OpenMP 4.5 の仕様にも同様に，USE_DEVICE_PTR という指示文が定義されており，GPU 上の配列がそのまま利用できるはずであるが，現時点では，Clang コンパイラー，XL コンパイラーともに，この指示文を利用することができない．

OpenMP 4.x において Unified Memory を利用するには，omp_target_associate_ptr という組み込み関数を利用して事前に Unified Memory で確保した配列のポインターをマップすることで, MAP 指示文によるコピーの生成を抑制できる．この組み込み関数は，C/C++で定義されているもので，Fortran からは直接は利用できない. 以下の C コードのような配列確保関数を用意しておくと便利である．

```
void* malloc_managed(size_t size){
  void* pP;
  cudaMallocManaged(&pP,size,cudaMemAttachGlobal);
  omp_target_associate_ptr(pP,pP,size,0,0);
  return pP;
}
void free_managed(void *pP){
  omp_target_disassociate_ptr(pP,0);
  cudaFree(pP);
}
```

このようにして確保しマップ済みになった配列は，次のような内積計算を行う C コードのように，直接利用することができるようになる．

```
double* a;
double* b;
double dot = 0.0;
a = malloc_managed(sizeof(double)*n);
b = malloc_managed(sizeof(double)*n);
#ifdef _OPNEACC
#pragma acc data deviceptr(a,b)
#pragma acc kernels
#pragma acc loop independent reduction(+:dot)
#else
#pragma omp target map(dot)
#pragma omp teams distribute parallel for reduction(+:dot)
#endif
for(i=0; i<n; i++){
  dot += a[i]*b[i];
}
```

なお, OpenACC では, PGI コンパイラーのオプションで, -ta=managed をつけることで, 全ての配列を Unified Memory として確保することができるが，これに相当する実装は今のところ OpenMP 4.x コンパイラーには存在しない．よって，Unified Memory として利用したい配列を明示的に確保してマップする必要がある．

2.4 OpenMP 4.x 実装の現時点での制限

いくつかのアプリケーションを OpenMP 4.5 を用いてポーティングを行ったところ，いくつかの制限があることが分かった．Clang コンパイラー，XL コンパイラーともに，前述の通り，USE_DEVICE_PTR 指示文はサポートされていない．また，OpenMP 3.0 で定義される task 指示文や，nowait depend 指示文を用いた非同期処理は，どちらのコンパイラーでもサポートされていない．XL コンパイラーにおいては，target 指示文内部（カーネル内部）で atomic 指示文を利用することができない. なお, Clang コンパイラーでは正しく atomic 命令を生成できる．

このように，OpenMP 4.x はコンパイラーの実装が OpenACC に比べて未成熟であり，今後の対応が待たれる．

3. アプリケーションを用いた性能評価

3.1 OpenPOWER と NVLink

NVLink は，NVIDIA 社の GPU 間を高速に接続するための新しいインターコネクトであり，同社の GPU である，Tesla P100 に初めて搭載された．また，NVLink によって，GPU のみならず, PCI Express の代わりに GPU と CPU の間を接続し，より高速なデータ転送を行うことが可能となった. NVLink は, 一本あたり片方向 20GB/s のリンクを, GPU あたり双方向に 4 組利用することができる．これらの 4 組のリンクをどのように接続するかは，システムによって自由度が与えられている．例えば，図 1 は IBM の OpenPOWER サーバー製品である, Power System S822LC for High Performance Computing (Minsky) [8]の NVLink の接続例を示す．CPU ソケットあたり 2 つの GPU を接続する構成では, 図 1 のように, 2 組ずつのリンクを用いて CPU と，2 つの GPU を相互結合することで，それぞれの間を双方向 80GB/s で接続している．これに対して現在の一般的な PC クラスター製品では，CPU と GPU の間は第三世代の PCI Express で接続するため，双方向で 32GB/s の転送速度となり，NVLink は 2.5 倍高速にデータ転送が行えることになる．（ただし，これは PCI Express のリンクが十分にある場合の速度であり，PCI Switch 等を利用してリンクを共有する場合は，さらに速度差が大きくなる．NVLink のもう一つの利点は，ネットワークインターフェース等，他の PCI Express 機器と独立したインターコネクトで GPU にアクセスできることである.）

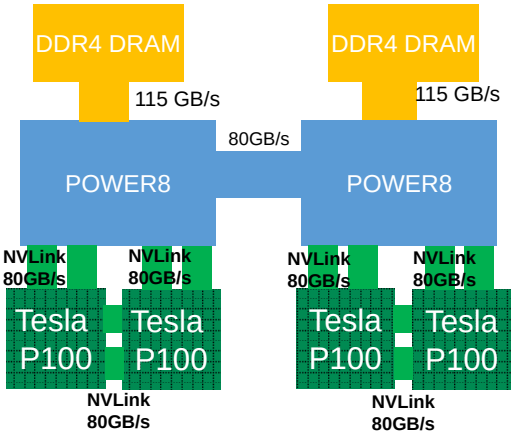

図 1　IBM Power System S822LC for High Performance Computing（Minsky）のシステム構成図とインターコネクトの転送速度（双方向）

表 1 性能評価に用いた Minsky の仕様概要

	IBM Power System S822LC for HPC (Minsky)
CPU	IBM POWER8
CPU コア数	10
CPU 周波数	2.86 GHz
CPU ソケット数	2
CPU メモリバンド幅	115 GB/s
GPU	NVIDIA Tesla P100
GPU 数	4
Interconnect	NVLink
コンパイラ	PGI Compiler 17.5 for Linux OpenPOWER IBM XL C/C++ Compiler version 13.1.5 Clang compiler (https://github.com/clang-ykt)
MPI	IBM Spectrum MPI 10.1.0

ここでは，表 1 に示す Minsky クラスターを用いて，ア

プリケーションの性能評価を行う. OpenACC と OpenMP 4.x それぞれの指示文を同一ソースコードに挿入し，性能を比較する．また，ここでは 1GPU あたり，1 つの MPI タスクを利用して実行するものとし，1 ノードあたり 4 GPU を用いるため，1 ノードあたり 4 MPI タスクを利用して実行する．このとき，それぞれの MPI タスクには，CUDA_VISIBLE_DEVICES 環境変数で使用する GPU を指定する．これによって，前述の omp_target_associate_ptr 関数でマップ先のデバイス番号を 0 番としても正しい GPU を選択できるようになる．

3.2 HPCG ベンチマーク

3.2.1 HPCG ベンチマーク概要

High Performance Conjugate Gradients (HPCG) Benchmark[9]は，LINPACK ベンチマークを補完する形で，新たなスーパーコンピューターの指標となることを目標として開発されたベンチマークプログラムであり，LINPACK ベンチマークによる Top500 List と同様，年 2 回順位が更新される．LINPACK ベンチマークでは，大規模な密行列を用いて直接法で方程式を解くが，実用的に大規模な密行列を扱うことは稀であり，また，計算機のピークパフォーマンスを測定するのと同じことになりつつある. それに対して，HPCG ベンチマークでは，疎行列を用い，共役勾配法（CG 法）を用いて方程式を解く問題を扱い，これは有限要素法などで広く使われる方法であり，より実践的なベンチマークである．また，こちらは，メモリバンド幅，計算能力，ネットワークバンド幅の総合的な計算機の能力を測るベンチマークである．

HPCG ベンチマークでは，CG 法の収束性を高めるための前処理としてマルチグリッド法が用いられており，この部分の処理が最も重い処理となっている．また，このマルチグリッド計算に，ガウスザイデル法が用いられている．ガウスザイデル法には計算順序に依存関係があるため単純に並列計算をすることは難しい．ガウスザイデル計算を GPU のような並列度の高い環境で並列化を行うためには，カラーリングによって計算順序を変えることで，スレッド間の値の書き換えによる不整合を防ぐ手法が広く用いられている．しかしながら，この手法は厳密には元のガウスザイデル計算と同じ計算をしていることにはならず，元の計算に比べて CG 法の収束回数が増えることが知られており，HPCG ベンチマークでは，収束回数の増加率によって，ベンチマークスコアを減点する仕組みとなっている．

3.2.2 HPCG ベンチマークのソースコード書き換え

HPCG ベンチマークを GPU を用いて並列化するにあたり，まずはソースコードの書き換えを行った．まず，ガウスザイデル計算を並列化するために，カラーリングによる計算順序の並び替えを行った．まずは，CPU 上で各色毎のループを OpenMP による SMP 並列化を行い，正常動作することを確認した．他のカーネルについても，SMP 並列化を行い，元の OpenMP 指示文から，OpenACC および，OpenMP 4 指示文へ書き換えることでカーネルをオフロードする．

また，疎行列の格納方式を，GPU で効率よくアクセスできるように書き換えを行う．元々の HPCG ベンチマークにおける疎行列の格納形式は CSR 形式であるが，各行の要素数を 0 を埋めることで同一にした ELL 形式をさらに転置した，転置 ELL 形式を用いた．例えば，式(1)のような疎行列は，図 2 に示すように格納される．

$$\begin{pmatrix} 1.5 & 0.2 & 0 & 0 \\ 0 & 1.0 & 3.2 & 0.5 \\ 0 & 0 & 2.5 & 0 \\ 0 & 1.1 & 0 & 2.2 \end{pmatrix} \quad (1)$$

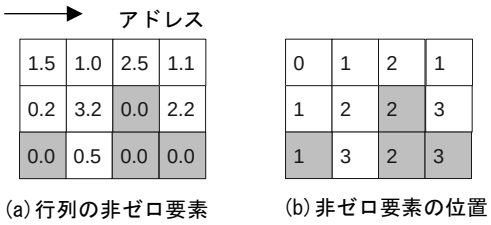

図 2　コアレスアクセスを考慮した転置 ELL 形式

GPU 上で使用する全てのデータは，Unified Memory として確保し，データの移動などの指示文は使用しない．

3.2.3 HPCG ベンチマークの性能評価

OpenACC および，OpenMP4 を用いて並列化した HPCG ベンチマークを Minsky1 ノードを用いて比較を行った．OpenMP4 は，XLC コンパイラーおよび，Clang コンパイラーのそれぞれを用いたものを比較した. また, 参考として，CUDA を用いて最適化をおこなったもの[10]（Unified Memory は不使用）とも比較を行った．結果を図 3 にまとめる．

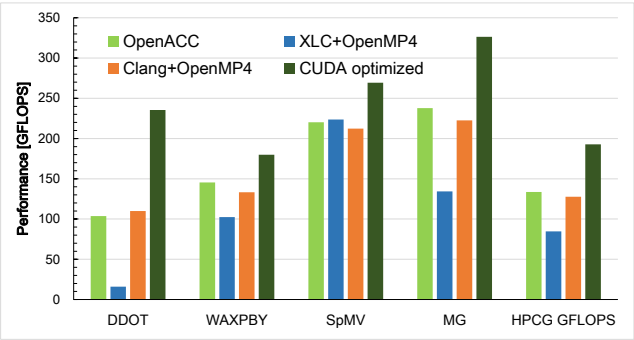

図 3　HPCG ベンチマークにおける各実装の性能比較. Minsky1 ノード，4GPU を使用（GPU あたりの問題サイズ：128x256x256）

図 3 では，4 つのカーネル(DDOT, WAXPBY, SpMV, MG)および HPCG ベンチマーク（バージョン 3）のスコアを比較している. XLC コンパイラによる結果が SpMV を除いてあまり良くないが，Clang コンパイラを利用した OpenMP4 の実装は，OpenACC による実装と遜色ない結果となった．CUDA による実装と比べて，OpenACC および OpenMP4 の

実装は，おおよそ 70%程度の性能となったが，Unified Memory のオーバーヘッドがあることと，少ない行数の指示文の追加のみであることを考慮すると，逆に高い生産性の割には十分な性能が得られているとも言える．

3.3 CoMD

3.3.1 CoMD 概要

CoMD[11]は, 分子動力学シミュレーションのためのリファレンス実装で, C 言語で書かれたシンプルな実装である．CoMD の実装は，反復計算によって三次元空間内の原子間の相互作用を解く N 体問題である．相互作用の計算には，Lennard-Jones（LJ）および embedded atom method（EAM）の 2 つのポテンシャルモデルを実装している．OpenMP による SMP 並列化および MPI による分散メモリ並列化がされている．分散メモリ並列化では，境界となる領域にある原子の属性値を隣接プロセス間で交換する．

CoMD では，三次元空間を離散化して三次元グリッドで表現し，各グリッド内に原子を割り当てる．このとき，図 4 に示すように，原子間力の及ぶ範囲（カットオフ半径）でグリッドを切っておけば，計算する相手の原子を自分のグリッドと隣接するグリッドの 27 個のグリッドから探索すれば良いことになる．

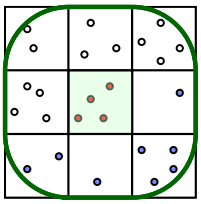

図 4　CoMD におけるカットオフ半径を考慮したグリッドサイズの決定

3.3.2 CoMD のソースコード書き換え

CoMD の SMP 並列化は，グリッド単位で並列化を行っている．すなわち，最外ループを並列化している．グリッド数が十分多ければ，単純に OpenMP の指示文を，OpenACC や OpenMP4 に書き換えることで, GPU 対応が可能である．まず，この一番単純な実装方法を，(a)とする．この並列化の方法では，原子の組み合わせについて，自分の原子のみに値を更新することで，異なるスレッドから同時に値を更新することを防いでいる．よって，2 個の原子の組について同じ計算を 2 度行うことになり非効率である．そこで，原子の組について一度しか計算せず，両方の原子について更新を行う方法を，atomic 指示文を用いて実装した．この実装を(b)とする．ただし，XL コンパイラーでは atomic 指示文が利用できないため，この方法は使用できない．

ここまでは非常に簡単に書き換えられるが，ここでは更なる最適化を試してみる．Yusof らによる CoMD の CUDA による最適化[12][13]では，さらに GPU に特化した最適化を施しているが，その中から，簡単に適用できる 2 つの手法を実装した．まずは，データアクセスを効率よく行うために，原子の属性値を Array of Structure（AoS）形式から，Structure of Arrays（SoA）形式に書き直した．具体的には，3 次元ベクトルを，3 つのベクトル成分の配列から，各成分毎の 3 つの配列に変換した．この実装を(c)とする．次に，最外ループをグリッドごとに並列化する手法では，各グリッドごとに持つ原子数にばらつきが生じロードバランスが悪化するため，グリッド単位ではなく原子単位で並列化を行う方法を試した．これには原子を 1 つの連続した配列として扱うため，原子へのポインタを持った配列と，所属するグリッドへのポインタを持った配列，の 2 つの配列を各スレッドが参照することで処理ができるようになる．この実装を(d)とする．

3.3.3 CoMD の性能評価

本報告では, EAM ポテンシャルモデルによる原子間の相互作用の計算カーネル（EAM force）について，OpenACC および OpenMP4 と Unified Memory を使用して，Minsky クラスター上で各実装について性能を比較した．図 5 にそれぞれの実装についての実行時間をまとめる．一番単純な実装(a)では OpenMP4 による実装が OpenACC の約半分の実行時間となっているが，atomic 指示文を利用した実装(b)では，OpenACC の実行時間が(a)から半減しているのに比べると OpenMP4 の実行時間の減少はだいぶ少ない．また，SoA 形式を利用した実装(c)の場合, OpenMP4 では逆に実行時間が増加してしまっている．つまり，あまりソースコードの最適化を行わない場合に OpenMP4 のコンパイラーによる最適化が非常に良く効くのに比べて，プログラマーが最適化を行った場合の効果は OpenACC の方が大きくなると言える．言い換えれば，生産性の観点では，OpenMP4 の方が有効であると言える．また，実装(d)では，OpenACC の実装はさらに高速化ができている．しかしながら，CUDA で最適化した実装（Unified Memory 不使用）に比べるとまだ倍近く遅い．それでも，比較的簡単なソースコード書き換えのみでここまで高速化できれば，実用上は十分なのではなかろうか．

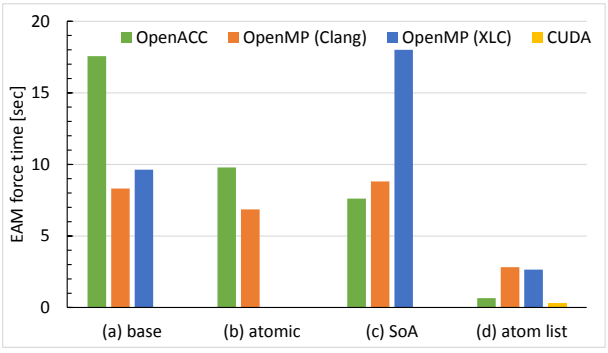

図 5　CoMD の EAM force カーネルの各実装の性能比較．Minsky1 ノード，4GPU を使用（問題サイズ：64x64x64）

4. おわりに

前回の報告では従来の PCI Express で GPU を接続する場合に比べて，より高速な NVLink の登場により，OpenACC と Unified Memory を組み合わせたプログラミング手法によって，アプリケーションを GPU に対応させる際に少ないソースコード書き換えで十分な性能が得られる可能性があることを示した．本報告では，さらに OpenMP 4.x を用いても同様に生産性と性能を両立できることを示すことができた．また，アプリケーションによっては，より単純な指示文の追加だけで OpenACC よりもいい性能が出せる場合があることが確認できた．

しかしながら，OpenMP 4.x の実装は現時点では未熟性であり，サポートされていない指示文があったり，十分に最適化できない場合がある可能性があることが分かった．特に，現時点では FORTRAN では Unified Memory が利用できないため，本報告の提案手法は C または C++での実装に限り有効な手法となっている．今後の実装において改善されるのを期待したい．

今後は，さらに多くのアプリケーションで性能を評価し，より特性を理解したい．また，実運用されるアプリケーションにも適用し，その評価を行いたい．また，NVLink2 を利用した場合にキャッシュコヒーレントな真の Unified Memory を使った場合の効果を試したい．さらに，GPU のメモリよりも大きいサイズの問題を扱う場合についても検証したい．

参考文献

[1] OpenACC More Science, Less Programming, https://www.openacc.org/
[2] OpenMP Application Programming Interface Version 4.5 November 2015, http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf
[3] Mark Harris, Unified Memory in CUDA 6, https://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/
[4] NVIDIA NVLINK HIGH-SPEED INTERCONNECT, http://www.nvidia.com/object/nvlink.html
[5] 土井　淳, NVLink における Unified Memory と OpenACC によるプログラミングの性能評価，第 159 回 HPC 研究会, 2017.
[6] Gheorghe-Teodor Bercea, Using OpenMP 4.5 in the CLANG/LLVM compiler toolchain, SUMMITDEV Workshop 2017.
[7] James Beyer, Jeff Larkin, Targeting GPUs with OpenMP 4.5 Device Directives, GPU Technology Conference 2016.
[8] IBM Power System S822LC for High Performance Computing, https://www.ibm.com/systems/jp-ja/power/hardware/s822lc-hpc/
[9] HPCG Benchmark, http://www.hpcg-benchmark.org/
[10] 土井 淳，奥野 伸吾, OpenPOWER クラスターにおける HPCG ベンチマークの最適化手法に関する考察，第 154 回 HPC 研究会, 2016.
[11] CoMD Proxy Application, http://www.exmatex.org/comd.html
[12] Jamal Mohd-Yusof, Nikolay Sakharnykh, Optimizing CoMD: A Molecular Dynamics Proxy Application Study, GPU Technology Conference 2014.
[13] GPU implementation of classical molecular dynamics proxy application, https://github.com/NVIDIA/CoMD-CUDA