Vol.2017-HPC-159 No.5
2017/4/17

NVLink における Unified Memory と OpenACC による プログラミングの性能評価

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多く登場し，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．プログラミングを複雑化する要因の一つとして，CPU と加速装置の間でデータをやり取りする必要があることが挙げられる．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，実用的な性能になる可能性がある．また，NVLink を用いることで，Unified Memory に OpenACC を組み合わせることで，最小のソースコード書き換えで，GPU を活用できることが期待される．本研究では，いくつかのアプリケーションについて，Unified Memory と OpenACC を組み合わせたプログラミングを行い，その性能評価を行った．

キーワード：NVLink, OpenACC, 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．このようなハイブリッドな計算機において計算を行うにはどこにデータがあるかを管理する必要があり，加速装置を利用して計算を行うためには，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スーパーコンピューターの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 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 を用いることで，比較的簡単に GPU による加速を得られる．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 GPU 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 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．

Unified Memory も，OpenACC 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．NVLink の登場によって，高速なデータ転送で 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．本研究では，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たい．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目標として，現時点でどの程度実用的であるかを検証するためにアプリケーションの性能評価を行う．

二章では，最近の GPU 計算機環境のまとめとして，NVLink，Unified Memory，OpenACC について概要を説明する．三章では，姫野ベンチマークを用いて，OpenACC および，Unified Memory を用いたプログラミング手法を順を追って説明する．四章では，姫野ベンチマークと，FIBER MINIAPP から CCS QCD を用いた性能評価について説明する．

2. 最近の GPU を利用した計算機環境

2.1 NVLink 概要

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[1]であり，同社の GPU である，Tesla P100[2]に初めて搭載された．また，NVLink によって，

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

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．OpenPOWER Foundation [3]では，POWER プロセッサーと GPU をこの NVLink で接続する設計のクラスターを提案しており, この規格の OpenPOWER クラスターでは, CPU と GPU の間でより効率的にデータのやり取りができるようになり，GPU プログラミングにおける性能に関わる大きな要素の一つを改善できる可能性がある．

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[4]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．）

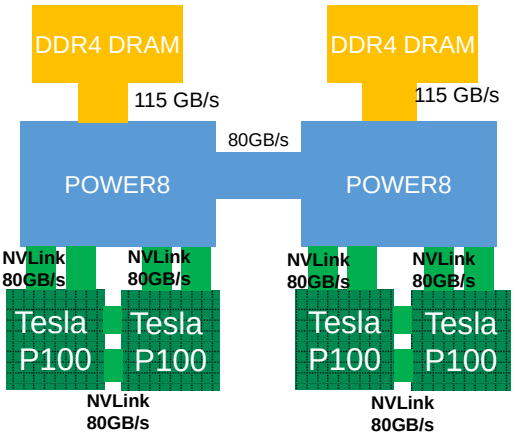

図 1　IBM Power System S822LC for High Performance Computing のシステム構成図とインターコネクトの転送速度（双方向）

2.2 Unified Memory 概要

Unified Memory[5]は，CUDA に実装されている仮想メモリの仕組みであり，ユーザープログラムから CPU と GPU で単一のメモリ空間を扱えるように, CPU-GPU 間のデータ転送を隠蔽した実装である．CUDA version 8 以降の（および，Pascal 世代以降の GPU における）Unified Memory の実装では，CPU と GPU それぞれにおいてページフォルトが発生した際に自動的にページ単位でデータ転送が実行される仕組みになっている．また，GPU に搭載されるメモリサイズよりも大きなメモリサイズを確保でき（CPU 側のメモリサイズ以下である必要があるが），GPU のメモリサイズに収まりきらないような問題を扱うことも可能となった．

プログラマーは，実データがどこにあるかを意識せずにプログラミングが行えるため，GPU へのコード移植が簡単になったり，プログラムコードの保守性が良くなったりといった利点が得られる．しかしながら，物理的にデータがどこにあるかを意識して書かれたプログラムに比べると，自動的に行われる CPU-GPU 間のデータ転送がボトルネックになる可能性がある．そのため，より転送速度の高速な NVLink が，Unified Memory を利用したプログラミングの生産性と性能を両立させる鍵になるかもしれない．

2.3 OpenACC 概要

OpenACC[6]は，GPU 等の加速装置をユーザープログラムから簡単に利用することができるように設計された，指示文ベースのプログラミング手法である．OpenMP による共有メモリ並列化と同じように，ループの前などに指示文を挿入することで, OpenACC に対応したコンパイラーが自動的に加速装置用のコードを生成する．また，CPU と加速装置の間のデータ転送や加速装置内のメモリ領域の確保等を指示文を用いて記述することができ，比較的少ないソースコード書き換えで，加速装置用にアプリケーションを移植できる．CUDA で最適化されたコードには及ばないことが，それでも OpenACC の指示文を挿入しただけでも，アプリケーションによっては十分な高速化が得られることが多い．

OpenACC では，CUDA よりは簡単とはいえ，データがどこにあるか，どのタイミングで転送をするか，を意識してプログラミングをする必要がある．プログラムが複雑になると, どのデータが転送されたかを考えるのも難しくなり，必要以上のデータ転送を行ってしまうと性能低下の原因にもつながる．そこで，Unified Memory を利用することで，OpenACC の複雑なデータ管理を自動化し, さらなるアプリケーションの生産性と性能の両立を目指す．

3. Unified Memory と OpenACC を利用したプログラミング

ここでは，姫野ベンチマーク[7]を例に，Unified Memory と OpenACC を利用したプログラミング手法について説明する．姫野ベンチマークについては今更詳細を説明するまでも無いほど，HPC 分野では広く用いられてきたベンチマークプログラムの一つであり，これまでにも，GPU を含む様々なシステムについての性能評価がなされてきている．ここでは，FORTRAN90+MPI 版の HimenoXP ベンチマークを利用して，(1)OpenACC を利用した GPU への移植，

(2)Unified Memory の利用による簡単化，の順に説明する．しかしながら，ここでは説明を簡単にするために，この順番であるが，実際のもう少し大規模なアプリケーションを移植する際には逆に最初から Unified Memory を用いてプログラミングを行うことで，移植をより簡単にできると考えている．

なお，本研究では，PGI FORTRAN Compiler 16.10 を利用した．

3.1 OpenACC による GPU プログラミング

OpenACC を FORTRAN から使う場合，前提として，コモンブロックに定義された変数や配列は利用できない．また，OpenACC2.0 の ROUTINE 指示文を利用する場合に，モジュールを用いた方が記述が簡単であるため，FORTRAN90 以上を利用する．

姫野ベンチマークは，ヤコビの反復法を計算する jacobi サブルーチンの時間計測を行うため，このサブルーチンを OpenACC を用いて GPU コードを生成する．次のように，2 つの 3 重ループに指示文を挿入し，それぞれ GPU で並列化する．このとき，3 重ループは独立に計算できるので，INDEPENDENT と，COLLAPSE(3)を指定し，最大限の並列度で計算できるようにする．また，gosa を集約計算するために，REDUCTION 指示文を用いている．

```
      gosa=0.0
!$ACC KERNELS
!$ACC LOOP INDEPENDENT COLLAPSE(3) PRIVATE(i,j,k,s0,ss)
REDUCTION(+:gosa)
      do k=2,kmax-1
         do j=2,jmax-1
            do i=2,imax-1
               s0=a(I,J,K,1)*p(I+1,J,K) &
                  !中略
               ss=(s0*a(I,J,K,4)-p(I,J,K))*bnd(I,J,K)
               gosa=gosa+ss*ss
               wrk2(I,J,K)=p(I,J,K)+omega*ss
            enddo
         enddo
      enddo
!$ACC END LOOP
!$ACC LOOP INDEPENDENT COLLAPSE(3) PRIVATE(i,j,k)
      do k=2,kmax-1
         do j=2,jmax-1
            do i=2,imax-1
              p(i,j,k)= wrk2(i,j,k)
            enddo
         enddo
      end do
!$ACC END LOOP
!$ACC END KERNELS
```

Tesla P100 用にコンパイルを行うには次のコンパイルオプションを指定する．

```
-acc -ta=tesla,cc60 -Minfo=accel
```

-Minfo オプションは必ずしも必要なものではないが，どのように GPU 向けのコードが生成されたかを確認できるため，つけておきたい．

これらの指示文を挿入した状態で，コンパイルを行うと，必要なデータ転送が自動的に挿入されて，一応 GPU で動くコードが生成できる．しかしながら，参照しかしない配列をカーネル実行後に CPU に戻すような転送等の不要なデータ転送が生成されてしまうことがあるので，次のように DATA 指示文を用いて，適切な処理を指定する．COPYIN は CPU から GPU へのデータ転送のみを行い，カーネル実行後に CPU に書き戻さない配列を示す．COPY はカーネル実行前に GPU にデータを転送し，実行後に CPU に値を戻す配列に指定する．ここでは使用していないが，COPYOUT は結果のみを CPU に戻す場合に用いる．CREATE は GPU 内のみで使用する配列を確保する．

```
!$ACC DATA COPYIN(a,b,c,wrk1,bnd) COPY(p) CREATE(wrk2)
  do loop=1,nn
     gosa=0.0
!$ACC KERNELS
  !省略
!$ACC END KERNELS
     call sendp(ndx,ndy,ndz)
     wgosa = gosa
     call mpi_allreduce(wgosa,gosa,1,mpi_real4, &
       mpi_sum,mpi_comm_world,ierr)
  enddo
!$ACC END DATA
```

jacobi サブルーチンでは，sendp サブルーチンで，配列 p の袖領域の交換を行っている．そこで，sendp サブルーチン内でも，MPI 通信で袖領域の通信ができるように，以下のように UPDATE 指示文によって配列 p の転送を指定する．UPDATE 指示文は，DATA 指示文で囲まれた領域内で，配列を更新したい場合に利用する．（実装１）

```
  if(ndz > 1) then
!$ACC UPDATE HOST(p(:,:,2),p(:,:,kmax-1))
     call sendp3()
!$ACC UPDATE DEVICE(p(:,:,1),p(:,:,kmax))
  end if
  if(ndy > 1) then
!$ACC UPDATE HOST(p(:,2,:),p(:,jmax-1,:))
     call sendp2()
!$ACC UPDATE DEVICE(p(:,1,:),p(:,jmax,:))
  end if
  if(ndx > 1) then
!$ACC UPDATE HOST(p(2,:,:),p(imax-1,:,:))
     call sendp1()
!$ACC UPDATE DEVICE(p(1,:,:),p(imax,:,:))
  end if
```

また，CUDA Aware MPI を用いる場合，これらのデータ転送は不要となるが，OpenACC では KERNELS 指示文の範囲外では配列は CPU 側のアドレスとして扱われるため，MPI に渡される配列は CPU のものとなりそのままではデータ転送が必要となってしまう．そこで，次のように HOST_DATA USE_DEVICE を使用することでその範囲内では，指定した配列は GPU 側のアドレスとして扱われるた

め，MPI に GPU 側の配列が渡されるようになりデータの転送が不要となる．（実装２）

```
! sendp1, sendp2, sendp3 サブルーチン内で
!$ACC HOST_DATA USE_DEVICE(p)
  call
mpi_irecv(p(1,1,kmax),1,ijvec,npz(2),1,mpi_comm_cart,ireq(3),ier
r)
  ! 省略
!$ACC END HOST_DATA
```

3.2 Unified Memory と OpenACC を組み合わせる

前節で説明したように, OpenACC では, DATA 指示文や, UPDATE 指示文によるデータの転送を明示的に行うことで, 無駄な転送を省いたプログラミングを行う. しかしながら, プログラムの構造が複雑になると，これらのデータ転送を管理するのが簡単ではなくなる場合もある．また，OpenACC で GPU への移植を行うときに，徐々にカーネル部分の移植を行う際には，最適なデータ管理を考えるのは難しい．そこで，DATA 指示文や UPDATA 指示文で転送の支持を明示的に行わずに，Unified Memory を用いて，必要なデータ転送は CUDA 任せにしてしまう方法が考えられる．

PGI コンパイラーの場合，簡単に Unified Memory を利用するには, コンパイルオプションを次のように変更すると，全ての配列は Unified Memory として扱われる．

```
-acc -ta=tesla,cc60,managed -Minfo=accel
```

このとき，明示的にデータの転送を記述する必要はなくなるため, 全ての DATA 指示文や UPDATE 指示文を省略できる．（実装３）

しかしながら，この方法では，性能については後述するが, 本来必要の無いデータ転送が生じることがあるようで, あくまでも，最初の移植時に試してみる程度の使い方をするのが良いようだ.

そこで, CUDA FORTRAN と OpenACC を組み合わせて, Unified Memory を利用する. CUDA FORTRAN では, Unified Memory として扱いたい変数や配列は，次のように属性として managed を付加することで定義できる．

```
real(4),dimension(:,:,:),allocatable,managed :: p
```

Unified Memory と OpenACC を組み合わせて使うには，次のようなコンパイルオプションを指定し，CUDA FORTRAN を有効にする．

```
-acc -Mcuda -ta=tesla,cc60 -Minfo=accel
```

GPU で利用する配列のみに，managed 属性を付加しておく．こうすることで，-ta=managed コンパイラオプションと同じように，managed 属性を持った配列に対しては，DATA 指示文や UPDATE 指示文を省略できる．この属性は，大本の配列の定義のみならず，サブルーチンの引数に渡される際にも同様に，managed 属性を指定する必要がある．（実装４）

また，実装３および４では配列のアドレスは，CPU と GPU で同一であるので，CUDA Aware MPI を利用して，データの転送を明示的に記述する必要も GPU のアドレスを取得する必要も無く，MPI 通信前後に指示文を追加する必要が無い.

このように，Unified Memory を組み合わせると，必要最小限の OpenACC 指示文の挿入だけで移植が可能となり，よりカーネル部分の移植に集中して作業できると思われる. また，必要に応じてデータ転送が行われるため，場合によっては，手動でデータを管理するよりも，性能が向上する可能性も期待できる.

4. Unified Memory と OpenACC によるアプリケーションの性能評価

ここでは，二つのベンチマークアプリケーションを用いて，OpenACC と，Unified Memory に OpenACC を組み合わせた場合の性能を比較する．また，比較のために，単一ノードではあるが，CPU-GPU 間の接続に，NVLink を用いる場合と，従来通りの PCI Express を用いる場合についても，性能を比較する．使用する計算機を表 1 にまとめる.

表 1 性能評価に用いた二つの計算機環境の比較

	x86 ノード	**IBM Power System S822LC for HPC (Minsky)**
CPU	Intel Xeon E5-2640v4	IBM POWER8
CPU コア数	10	10
CPU 周波数	2.4 GHz	2.86 GHz
CPU ソケット数	2	2
CPU メモリバンド幅	68.3 GB/s	115 GB/s
GPU	NVIDIA Tesla P100	NVIDIA Tesla P100
GPU 数	4	4
Interconnect	PCIe gen.3	NVLink
コンパイラ	PGI FORTRAN 16.10	PGI FORTRAN 16.10 for Linux OpenPOWER
MPI	OpenMPI 1.10.2	IBM Spectrum MPI 10.1.0

また，ここでは OpenACC アプリケーションは，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.

4.1 姫野ベンチマーク

前章で説明したとおり，姫野ベンチマークは，FORTRAN90+MPI 版の HimenoXP ベンチマークを利用した. 問題サイズとして，L および XL を利用して実行した場合

の，実装１～４について性能を比較した．

まず，単一ノードを用いて，それぞれの実装について性能測定を行った．図 2 に L サイズを用いた場合，図 3 に XL サイズを用いた場合について実効性能値をまとめる．実装１と実装２では，PCI Express 接続の場合と NVLink 接続の場合で，CPU-GPU 間の転送速度の差が，性能の差に表れているのが分かる．また，NVLink の結果では，実装２で CUDA Aware MPI の効果が見られオーバーヘッド低減による性能向上が見られる．実装３と４を比較すると，-ta=managed コンパイルオプションのみに頼るよりも，個別に属性を指定した実装４の方が良い性能が出ている．Unified Memory を使わない場合と比べると，XL サイズの場合でやや良い性能が得られたが，L サイズでは性能が負けている．Unified Memory を利用する場合，何らかのオーバーヘッドが CUDA 内部で生じており，問題サイズがある程度大きくないと性能が落ちるものと思われる．それでも，NVLink を利用することで，少ないコード書き換えで十分な性能が得られていると言える．

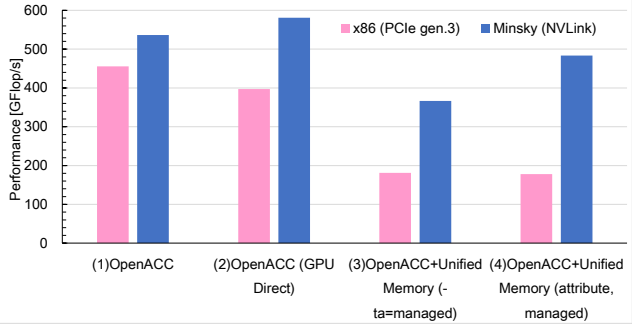

図 2 姫野ベンチマーク（L サイズ）を１ノードで実行したときの性能比較

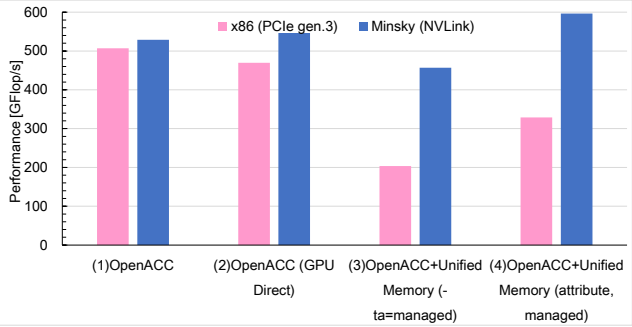

図 3 姫野ベンチマーク（XL サイズ）を１ノードで実行したときの性能比較

次に，NVLink を用いて，複数ノードで姫野ベンチマークを実行した．同様に，図 4 に L サイズを用いた場合，図 5 に XL サイズを用いた場合について実効性能値をまとめる．単一ノードで実行した場合と同様に，L サイズでは，Unified Memory による実装は，従来手法に比べて性能が劣っている．また，使用するノード数が増加すると，その傾向が顕著になるため，Strong Scaling もあまり良くないことが分かる．一方 XL サイズでは，実装４は比較的良好な性能が得られている．

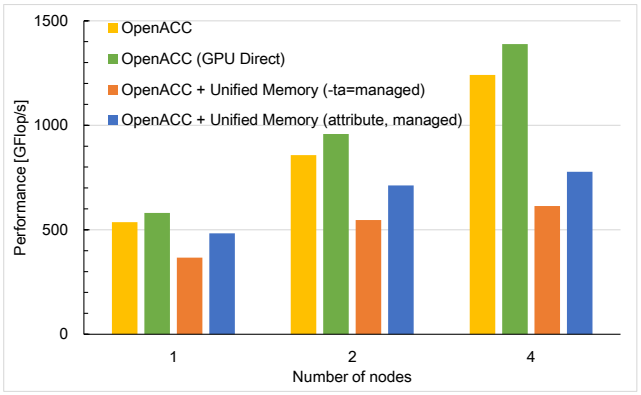

図 4 姫野ベンチマーク（L サイズ）を OpenPOWER クラスター（NVLink を使用）で１～４ノードで実行したときの性能比較

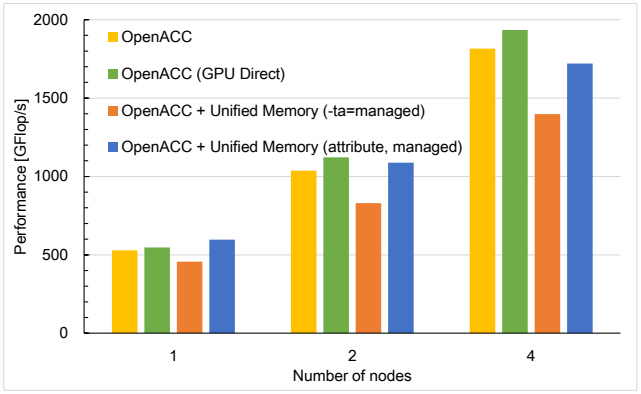

図 5 姫野ベンチマーク（XL サイズ）を OpenPOWER クラスター（NVLink を使用）で１～４ノードで実行したときの性能比較

4.2 CCS QCD

CCS QCD は，FIBER MINIAPP[8][9]の中の一つのベンチマークアプリケーションである．FIBER MINIAPP は，理化学研究所によって開発されたベンチマークプログラム集で，京コンピューター等の国内のスーパーコンピューター等で良く利用されるアプリケーションの計算カーネル部分を抜き出し，性能評価に利用できる形で提供されている．

本研究では，この中から，既に OpenACC による GPU 対応コードが用意されている，CCS QCD を用いて，性能評価を行う．CCS QCD は，格子 QCD の実装の一つで，FIBER MINIAPP における CCS QCD は，格子 QCD シミュレーションで多用される線型方程式ソルバーを BiCGStab 法による反復解法で求める部分を切り出したものとなっている．

Unified Memory を利用した実装は，GPU で使用する全ての配列の属性を，managed に変更し，全てのデータ転送指示文をコメントアウトすることで実装した．

まず，単一ノードを用いて，それぞれの実装について性能測定を行った．図 6 に class 1 の問題サイズを用いた場合，図 7 に class 2 の問題サイズを用いた場合についてそ

れぞれの実効性能をまとめる．

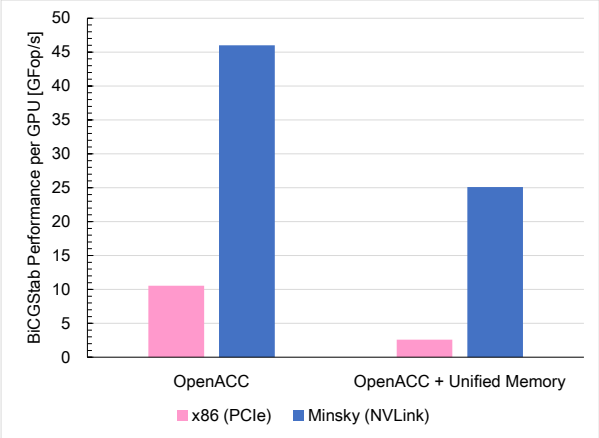

図 6 CCS QCD ベンチマーク（class 1, 8x8x8x32）を 1 ノードで実行したときの性能比較

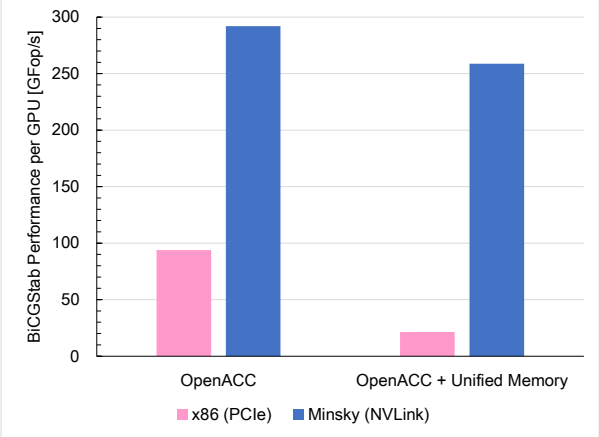

図 7 CCS QCD ベンチマーク（class 2, 32x32x32x32）を 1 ノードで実行したときの性能比較

姫野ベンチマークの結果と比べると，PCI Express 接続の場合と NVLink 接続の場合で，性能差の開きが増大した．アプリケーションごとの特性によって，NVLink の効果の大きさが変わり，格子 QCD の場合，交換する袖領域のデータ量が比較的多いために，性能差も大きくなったと考えられる．Unified Memory を使った実装では，姫野ベンチマークと同様に，問題サイズが大きいほうが良い性能が出る傾向にあることがわかる．class 2 では，元の OpenACC 実装に近い性能が得られた．

次に，NVLink を用いて，複数ノードで性能測定を行った．図 8 に class 2 の問題サイズを用いた場合，図 9 に class 3 の問題サイズを用いた場合についての実効性能をまとめる．

class 2 では，Unified Memory による実装は若干性能が劣っているが，そもそもの問題サイズがそれほど大きくないため元の OpenACC の実装で 8 ノードで性能が下がっているのが分かる．一方，class 3 では，Unified Memory による実装の方がやや高い性能を示した．

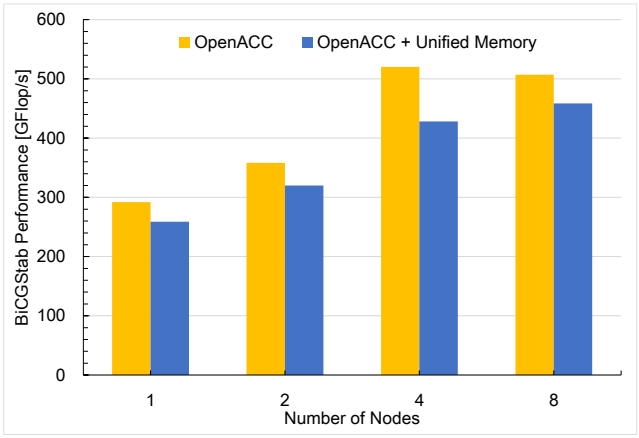

図 8 CCS QCD ベンチマーク（class 2, 32x32x32x32）を OpenPOWER クラスター（NVLink を使用）で 1 ～ 8 ノードで実行したときの性能比較

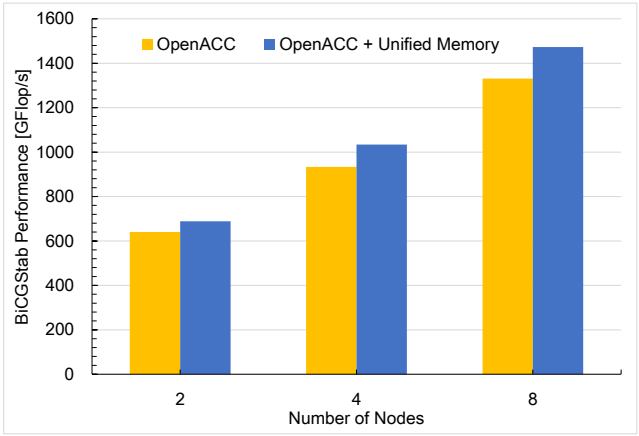

図 9 CCS QCD ベンチマーク（class 3, 64x64x64x32）を OpenPOWER クラスター（NVLink を使用）で 2 ～ 8 ノードで実行したときの性能比較

5. おわりに

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，今までのプログラミング手法とは別の，もっと簡単な手法を使っても，GPU を活かすことができる可能性が出てきた．本研究で提案した，OpenACC と Unified Memory を組み合わせたプログラミング手法は，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が分かった．特に，プログラマーがデータの管理をする必要が無くなり，カーネルコードの移植に集中できるのが大きい．

しかしながら，現時点では NVLink をもってしても，Unified Memory のオーバーヘッドは無視できないということも分かった．問題サイズが十分に大きければ，このオーバーヘッドは無視できるが，strong scaling を考慮すると性能の低下は避けられない．OpenACC と Unified Memory の組み合わせは，アプリケーションの種類と問題サイズを選

べば，従来のプログラミング手法と遜色ない性能，あるいは，良い性能を引き出せる．

次世代の NVLink である，NVLink2 の登場により，このプログラミング手法は，さらに実用的なものになると予想できる．NVLink2 ではバンド幅が増大するとともに，CPU-GPU 間でキャッシュコヒーレントになり，現在の Unified Memory の実装よりも，より低いオーバーヘッドで扱えるようになると予想できる．

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．今回は FORTRAN で記述されたアプリケーションのみを対象としたが，C/C++で記述されたものについても試していきたい. また, OpenACC に続いて，OpenMP 4.x による GPU プログラミングについても同様に性能を評価していきたい．

参考文献

[1] NVIDIA NVLINK 高速インターコネクト, http://www.nvidia.co.jp/object/nvlink-jp.html
[2] Tesla P100 最先端のデータセンターアクセラレータ, http://www.nvidia.co.jp/object/tesla-p100-jp.html
[3] OpenPOWER foundation, http://openpowerfoundation.org/
[4] IBM Power System S822LC for High Performance Computing, https://www.ibm.com/systems/jp-ja/power/hardware/s822lc-hpc/
[5] Mark Harris, CUDA 8 Features Revealed, https://devblogs.nvidia.com/parallelforall/cuda-8-features-revealed/, 2016
[6] OpenACC, http://www.openacc.org/
[7] 姫野ベンチマーク, http://accc.riken.jp/supercom/himenobmt/
[8] 小村幸浩，鈴木惣一朗，三上和徳，滝澤真一朗，松田元彦，丸山直也, Fiber ミニアプリの性能評価, SWoPP 2014 workshop, 2014.
[9] FIBER MINIAPP SUITE, http://fiber-miniapp.github.io/