Vol.2016-HPC-156 No.14
2016/9/16

Strong Scaling を考慮した CUDA プログラミング手法についての考察

土井淳[†1]

概要：GPU のようなアクセラレーターを利用したクラスターは HPC 分野において大きな勢力となりつつある．これらのアクセラレーターを利用するためには，ホスト CPU から，アクセラレーターを制御する必要があり，ホスト側の CPU においてもそれなりの負荷が発生する．CUDA プログラミングにおいて GPU とのデータ転送やカーネル実行は非同期で行うことが可能であり，その際ホスト側の CPU の処理はブロックされない．しかしながら，データ転送やカーネル実行の際に，CPU 側で準備のオーバーヘッドが生じ，この部分は非同期ではない．一般的な CUDA プログラミングの最適化手法として，これらのオーバーヘッドを相対的に無視するために，できるだけ大きな問題を GPU で処理することが推奨されている．しかしながら，問題サイズを大きくできる状況は限られており，複数の GPU や複数ノードを利用した並列計算においては，特に strong scaling を考えると問題サイズは並列数に反比例して小さくなり，相対的に CUDA のオーバーヘッドが増大し性能低下の原因となる．また，将来的に CPU-GPU 間の転送速度が速くなったり，GPU の処理速度が上がった場合にも，同様に相対的なオーバーヘッドは増大してしまう．本発表では，CUDA のオーバーヘッドを隠蔽することで，strong scaling 時の性能低下を抑制する手法について検討し，格子 QCD を例に OpenPOWER クラスター上での性能比較を行った結果を紹介する．

A Study of a Strong Scale Aware CUDA Programming Technique

JUN DOI[†1]

Keywords: GPU, GPGPU, CUDA, Strong scaling, Lattice QCD

1. はじめに

GPU に代表されるようなアクセラレータ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システムにおいて，アクセラレータにどのような計算をどのくらいの規模で行わせるかによって得られる性能が大きく変わる．一般的には，連続的に大規模な処理がある場合に良い性能を得ることができるとされ，それはアクセラレータとホストの間でデータのやり取りや，アクセラレータを制御するためのオーバーヘッド等の，計算そのものに追加される処理時間の影響をなるべく小さくするためである．データ転送にかかる処理時間については，計算とデータ転送を重ね合わせることで隠蔽する手法は数多く提案されているが，オーバーヘッドに関しては簡単には解決できない問題である．

GPU は，非同期的に動作させることができ，CPU をブロックせずに処理を行うことが可能であるが，GPU の制御にかかるオーバーヘッドについては，CPU はブロックされてしまう処理であり，CPU の性能に依存するものである．オーバーヘッドを無視するためには問題サイズを十分大きくとればいいが，いつも問題サイズを大きくできる状況であるとは限らない．特に分散メモリ並列化を考えると，問題サイズを分割していくと，GPU あたりの問題サイズは小さくなり，オーバーヘッドがアプリケーションの性能に与える影響は無視できなくなってしまう．

本研究は，特に分散メモリ並列化における Strong Scaling 性能について，CUDA プログラミングにおけるオーバーヘッドを抑えて，性能低下を改善することに着目した．CPU と GPU 間の同期を減らすことで，CUDA オーバーヘッドを演算処理に重ね合わせて隠蔽する手法について提案する．

2. CUDA プログラミングにおけるオーバーヘッド

2.1 CUDA プログラミング概要

CUDA(Compute Unified Device Architecture)は, NVIDIA 社が提供する GPU を用いて並列計算を行うための統合開発環境[1]であり, これを利用することで比較的容易に GPU による超並列計算の恩恵を受けることができるようになる．CUDA は，C++言語を拡張した開発環境であり，C/C++プログラムから，GPU 向けのコードを呼び出して利用する．CUDA では, GPU の制御やリソースの管理などは API 内部で処理され，プログラマーが特別な処理を記述する必要がなく，プログラマーは実行したい計算の実装に集中することが可能である．

CUDA によるプログラミング手順は，おおよそ次のような流れである．

(1) GPU に配列を確保する
(2) GPU の配列に利用したいデータを転送する
(3) GPU 上でカーネルを実行し計算を行う

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

(4) GPU の配列に保存された計算結果をホスト側に転送する

GPU を用いて計算を行うには，GPU のメモリ上に必要なデータを用意する必要があり，また，ホスト側で計算結果が必要な場合は，GPU のメモリ上のデータは直接山頂で着ないので，データを転送する必要がある．

したがって，一般的に，CPU 上での計算時間＞データ転送時間+GPU 上での計算時間，となるような処理でなければ GPU を用いることによる性能向上が得られない．したがって，できるだけ大きな問題を GPU 上で解いたり，データ転送を最小限に抑えるか，計算とデータ転送を重ね合わせるような手法が用いられる．

2.2 CUDA オーバーヘッド

CUDA によって，GPU の制御やリソースの管理はプログラマーが気にする必要はなくなったが，これらの処理は API 内部でホスト側の CPU を用いて処理される．すなわち，CUDA によって GPU で何らかの処理をさせようと思うと，必要に応じて CPU にも負荷がかかることになる．ここでは，CUDA の API やカーネル呼び出しについて CPU にかかる負荷のことを CUDA オーバーヘッドと呼ぶ．CUDA プログラミングにおいて，特に注目すべき主な CUDA オーバーヘッドには次のようなものがある．

(1) カーネルオーバーヘッド
(2) データ転送オーバーヘッド
(3) ストリーム制御オーバーヘッド

カーネルオーバーヘッドは，GPU においてカーネルを実行する前に，カーネルコードや引数を転送したり，GPU の制御をするための処理時間であり，呼び出すカーネルによってオーバーヘッドの大小がある．特に多くの引数を渡す場合，オーバーヘッドは大きくなる．データ転送オーバーヘッドは，cudaMemcpy 関数によって CPU と GPU 間でデータを転送するための準備にかかる処理時間である．ストリーム制御オーバーヘッドは，非同期処理を制御するための API 群の呼び出しコストである．

一般的にこれらのオーバーヘッドは，データ転送や計算処理に比べると小さく，あるいは，多くの CUDA プログラミングの教科書等ではこれらのオーバーヘッドを無視できるくらい小さくなるように問題サイズを大きくすることを推奨している．しかしながら，必ずしもオーバーヘッドを無視できるわけではなく，CUDA オーバーヘッドが性能低下の原因となりうる場合も多くある．

図 1 および図 2 に，同じ CUDA プログラムを大きさの違う問題を用いて実行した場合の，ビジュアルプロファイラー[2]による実行時間のイメージを示す. 図 1 では比較的小さいな問題サイズを用いており，CUDA オーバーヘッドによる経過時間が，実際のカーネル実行時間やデータ転送時間に比べて無視できないほど大きいことが分かる. また，本来このプログラムではデータ転送とカーネル実行は非同期に重ね合わせることができるが，CUDA オーバーヘッドが大きいために，データ転送が終わってもまだカーネル実行が始まっていない．

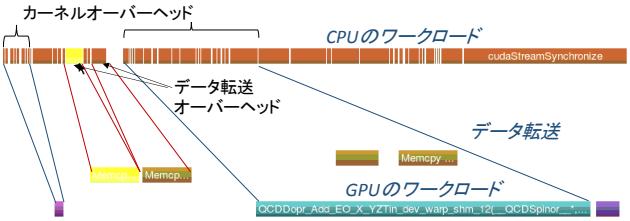

図 1 問題サイズが小さい場合の CUDA オーバーヘッドと実際のワークロードの処理時間の関係

一方，図 2 では，比較的大きな問題サイズを用いているため，CUDA オーバーヘッドは無視できる程度に小さくなっている．（なお，横方向のタイムスケールは図 1 と図 2 では同一ではないことに注意.）また，図 2 ではデータ転送完了前にカーネル実行がスタートしており，意図した通りに，データ転送とカーネル実行を重ね合わせることができている．

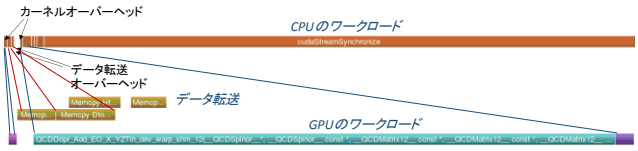

図 2 問題サイズが大きい場合の CUDA オーバーヘッドと実際のワークロードの処理時間の関係

このように，非同期処理を用いていても，CUDA オーバーヘッドが大きければ，その分 CPU がブロックされ，実は非同期的には実行されていなかったということもある．

2.3 相対的に大きくなる CUDA オーバーヘッド

CUDA オーバーヘッドを無視することができない状況の一つとして，複数の GPU やクラスター上の複数のノードを用いた分散並列計算を行う場合がある．特に問題サイズを固定して並列度を上げる Strong Scaling を考えると，並列数を増やすことで GPU あたりの問題サイズは小さくなり，処理時間が減少することで，図 3 に示すようにカーネルオーバーヘッドとデータ転送オーバーヘッドは，処理時間に対して相対的に増大していく．また，GPU におけるカーネルの実行時間は，GPU の性能に依存し，将来的に GPU の性能が上がると，実行時間が短くなり，図 4 のように相対的にカーネルオーバーヘッドは増大する．CUDA オーバーヘッドは CPU の性能に依存し，また，近年の GPU の性能向上のスピードは CPU の性能向上よりも速いため，1 世代新しい GPU が登場すると，CUDA オーバーヘッドは相対的に大きくなることになる．

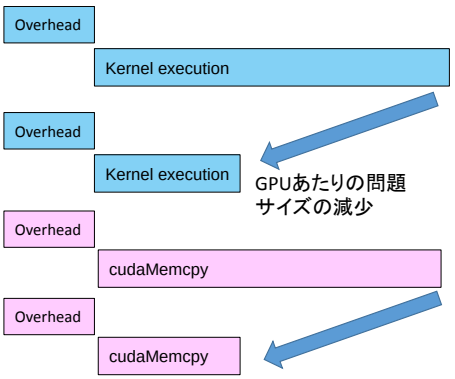

図 3　Strong Scaling によるカーネルオーバーヘッドとデータ転送オーバーヘッドの相対的な増加

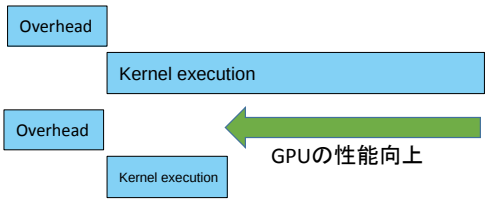

図 4　GPU の性能向上による相対的なカーネルオーバーヘッドの増加

GPU の性能向上と同様に，CPU-GPU 間のリンク速度の向上によっても，図 5 のように相対的にデータ転送オーバーヘッドは増大してしまう．次世代インターコネクトである NVLink[3]は，従来の PCI Express 接続の場合よりも 2 倍以上高速でありその分相対的なオーバーヘッドは大きくなる．

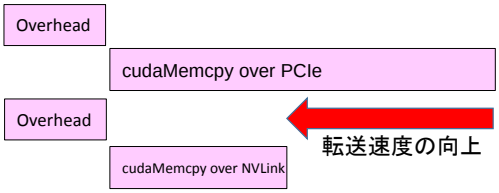

図 5　CPU-GPU 間のリンク速度の向上による相対的なデータ転送オーバーヘッドの増加

3. Strong Scaling を考慮した CUDA プログラミング

3.1 CUDA ストリームを利用した非同期プログラミングによるオーバーヘッドの隠蔽

逐次的に GPU に処理を行わせる場合，図 6 に示すように，CPU は CUDA オーバーヘッドを負って処理を GPU に投入すると，GPU が処理を完了するまで待って（GPU の処理が完了するまで CPU はブロックされる）から，次の処理を投入する．このとき CUDA オーバーヘッドも全体の実行時間に加算されている．

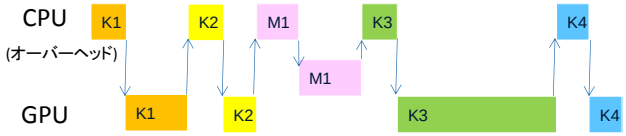

図 6　GPU による逐次的な処理の実行例

これに対して，図 7 では CUDA ストリームを用いることで，GPU と CPU を非同期的に処理している．CUDA ストリームを用いて，カーネル実行やデータ転送をストリームのキューに積むことで，前の処理が終わり次第，キューから次の処理を取り出して自動的に実行することができるので，CPU 側ではカーネル実行の完了を待つ必要が無く，処理がブロックされない．そのため，別のカーネル実行中に，CUDA オーバーヘッドを前払いして，次の処理の“予約”をすることが可能である．こうして，CUDA オーバーヘッドを隠蔽することができる．そこで，可能な限り CUDA ストリームを利用して非同期的に処理を行うことで，CUDA オーバーヘッドによる性能低下を小さくできると考えられる．

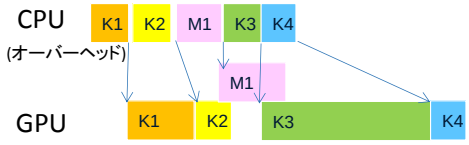

図 7　GPU による CUDA ストリームを利用した非同期的な処理の実行例

ただし，図 7 の例は，それぞれの処理の依存関係が比較的単純な例であり，MPI 通信が絡む場合等のように CPU と GPU の間で同期を取る必要がある場合には，ストリームで実行中の全ての処理の完了を待つ必要がある．これは，現時点（CUDA7.5 までの時点）で，CUDA ストリームを用いて GPU と CPU の処理の同期をとる仕組みが用意されていないからである．なお，GPU 内の複数のストリーム間の待ち合わせや，同一ノード内の GPU 間の待ち合わせには，CUDA イベント API を用いることで CPU をブロックすることなく処理を行うことができる[4]．図 8 の例では，K2 と M1 は 1 つ目のストリームで，それ以外は 2 つ目のストリームで実行され，K4 は K3 が完了し，かつ M1 のデータ転送が終わってから実行される．この場合，2 つのストリームの待ち合わせのために，CUDA イベントが用いられるが，図 8 に E1 と示したようにイベントのための API 呼び出しにも別のオーバーヘッドが発生する．

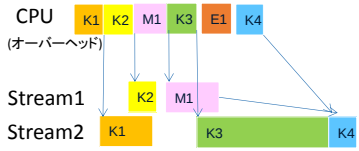

図 8　複数の CUDA ストリーム間の待ち合わせを含む場合の非同期処理の例

3.2 CPU-GPU 間の同期が必要な場合のオーバーヘッド隠蔽方法の提案

CUDA ストリームを使用するとき，CPU と GPU の間で同期を取る必要があるような処理としては次のようなものが考えられる．

(1) MPI を用いて他のプロセスとデータ通信を行う必要がある場合（GPU Aware MPI[5]を使用する場合でも）
(2) リダクション演算のように複数の GPU の結果を CPU で集約する必要のある場合
(3) CPU にも一部の計算処理を分配する場合

このような場合，一旦処理を CPU に戻すとき，CUDA ストリーム内の処理が全て完了するまで CPU はブロックされ，また，同期を取って再び GPU に処理を任せるには，また CUDA ストリームのキューに処理を積みなおすことになる．よって，このような同期が必要な処理が途中にあると，CUDA オーバーヘッドの隠蔽効率が下がってしまう原因になる．

図 9 に示す例は，MPI 通信を行うために CPU と GPU で同期を取る必要がある場合の例である．この例では，K1 で計算した結果を MPI を用いて別のプロセスに送り，また，別のプロセスから送られてきた結果を K3 で利用する．K2 は通信には依存しない計算で，また，K2 の結果を K3 も利用するとする．ここでは，データ転送（M1，M2）とカーネル（K1，K2，K3）は別のストリームを利用しているため，K1 から分岐する部分と K3 に合流する部分に CUDA イベント（E1，E2）を利用している．

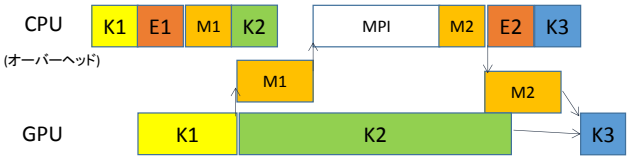

図 9　MPI 通信によって CPU-GPU 間の同期が必要となる場合の処理の例

この例では，M2 による転送を，MPI 通信が完了するまで CUDA ストリームのキューに積むことができないため，M2 に依存する他の処理も前もってキューに積むことができないため，このように一旦ストリームを止めなければいけない．

そこで，我々は，GPU 側で待ち合わせに使う小さなカーネルを用意し，CPU 側からこのカーネルを停止させることで CPU-GPU 間の同期を CUDA ストリームを止めずに行う手法を提案する．

待ち合わせに使用するカーネルは，1GPU スレッドのみを利用した単純なスピンループを持つカーネルで，アトミック関数を使用して，GPU 上のメモリに用意されたカウンターの値を監視し，目的の値になると，ループを抜けてカーネルを終了するだけのものである．

また，CPU 側からこのカーネルを停止させるには，別の 1 GPU スレッドからなる停止カーネルを実行する．このカーネルはアトミック関数を使って，GPU メモリ上のカウンタの値を目的の値に変更するだけのものである．

また，CPU 上では，GPU への処理の管理を行う主スレッドの他に，待ち合わせに使用するためのスレッドを用意する．この追加のスレッドが主スレッドに代わって，GPU との待ち合わせ時にブロックされることで，主スレッドはブロックされずに CUDA ストリームに処理を追加し続けることができ，CUDA オーバーヘッドを他の処理に重ね合わせて隠蔽し続けることが可能となる．

図 10 に，図 9 の MPI 通信がある場合の例について，提案手法で実装した場合の処理の流れを示す．W1 は，待ち合わせのためのカーネルで，K2 の実行が終わるとこのストリームは準備が整うまでブロックされる．そのため，続くカーネル K3 はこの時点でキューに追加することが可能となった．追加された CPU2 で実行されるスレッドでは，M1 のデータ転送を待った後，MPI 通信を行い，M2 によるデータ転送と続いて同一ストリームに待ち合わせのカーネルを停止させるためのカーネル S1 を追加する．データ転送 M2 が終わった直後に S1 が実行され，W1 の実行が停止されると K3 が実行される仕組みである．そのため図 9 で必要であった E2 は不要となった．この仕組みにより，CPU1 はブロックされることなく，より多くの CUDA オーバーヘッドを前払いして隠蔽できるようになる．

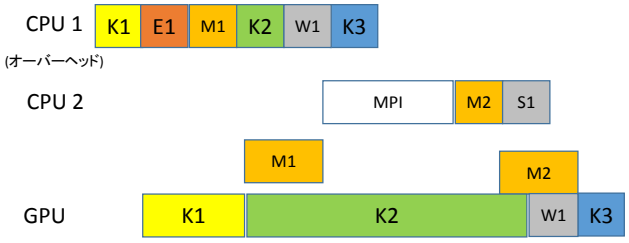

図 10　提案手法によって CPU-GPU 間の同期が必要となる場合でもストリームを止めずに処理を続けられるようにする例

図 9 と図 10 を比較した場合，この範囲の処理時間だけを比較すると大きな違いは無いが，この前後の処理をあわせると，大きな違いが出てくる．

次の章では，提案手法を，格子 QCD の実装に適用した場合の効果について検証する．

4. 格子 QCD を用いた性能評価

4.1 格子 QCD 概要

格子 QCD（Quantum Chromodynamics）は強い力の場の理論を離散化してコンピュータシミュレーションで解ける形にしたもので，世界中の多くのスパコンシステムにおいて実行されるアプリケーションの一つである. 格子 QCD は，様々な物理現象を再現したり理論を証明したりするのに役立っており[6][7]，計算機能力の向上により，未解明や未発見の事象についてのコンピューター上での再現が望まれている．

格子 QCD は 4 次元時空間を格子上に離散化した問題として扱い，格子点間の相互作用を用いて CG 法などを用いて線形方程式を解く．相互作用を計算する方法はいくつかあるがここでは良く使われる Wilson-Dirac 演算子を用いる. Wilson-Dirac 演算子は，式 1 に示されるような形で，スピノル場について，ゲージ場の影響を，隣接格子点の 4x3 のスピノル行列と 3x3 のゲージ行列を乗じることで求める 9 点ステンシル計算である．

$$D(n)=\delta(n)-\kappa\cdot\sum_{\mu=1}^{4}\left\{(1-\gamma_\mu)U_\mu(n)\delta(n+\hat{\mu})+(1+\gamma_\mu)U_\mu^{\dagger}(n-\hat{\mu})\delta(n-\hat{\mu})\right\} \quad (1)$$

また，SU(3)対象性を利用して，4x3 のスピノル行列は，半分のハーフスピノルと呼ばれる 2x3 行列に変換することでゲージ行列の乗算と，分散メモリ並列化時の境界部分の交換時のデータ転送を半分にできることが知られている．式 2 は X 軸の正の方向の例を示している．

$$(1-\gamma_1)U_1(n)\delta(n+\hat{1},m)=\begin{pmatrix}U_1(n)\cdot(s_1+i\cdot s_4)\\U_1(n)\cdot(s_2+i\cdot s_3)\\-i\cdot U_1(n)\cdot(s_2+i\cdot s_3)\\-i\cdot U_1(n)\cdot(s_1+i\cdot s_4)\end{pmatrix}=\begin{pmatrix}U_1(n)\cdot h_1\\U_1(n)\cdot h_2\\-i\cdot U_1(n)\cdot h_2\\-i\cdot U_1(n)\cdot h_1\end{pmatrix} \quad (2)$$

Wilson-Dirac 演算子の分散メモリ並列化では，4 次元格子を各プロセスに分割して処理を行う．分割された格子の境界部分では，互いのプロセス間で境界部分のデータを交換する必要がある．GPU による実装では，最内ループのメモリアクセスの最適化の観点から，X 軸方向は分割せずに，残りの，Y，Z，T 軸方向についてプロセス間で分割する．

4.2 CUDA オーバーヘッドを隠蔽した Wilson-Dirac 演算子の実装

Wilson-Dirac 演算子では，Y，Z，T 軸方向の境界部分について，MPI を用いて隣接プロセス同士でデータの交換を行う．そのため，本提案手法を用いて，CUDA ストリームを停止せずに MPI 通信を実現することで，比較的小さな格子データについても CUDA オーバーヘッドを隠蔽することで性能を向上できると考えられる．

図 11 に，提案手法を用いて Wilson-Dirac 演算子を GPU 用に実装したときの様子を示す．

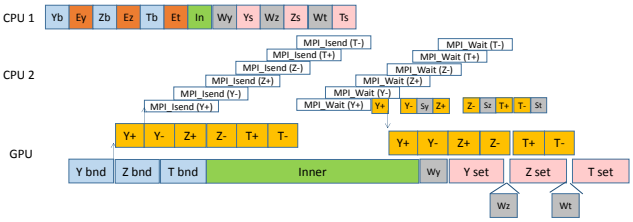

図 11　提案手法による Wilson-Dirac 演算子の実装

GPU 上で動作するカーネルは，全て同一の CUDA ストリーム上で実行される．Y bnd 等と示されている部分は，境界部分のハーフスピノルを計算して送信バッファに保存するカーネルで，送信バッファのデータは，データ転送用の CUDA ストリームを用いて CPU 側へ転送される．そのため CUDA イベントを使ってストリーム処理を分岐させている．CPU2 では，MPI_Isend/MPI_Irecv を用いて非同期で隣接プロセスに境界部分のデータを送受信し，受信した境界部分のデータを GPU に転送する．続けて，待ち合わせカーネルを停止させるためのカーネルを呼び出す．Y set 等と示されている部分は，受け取ったデータを使って境界部分のスピノルを更新するカーネルで，Inner（内部の格子点を計算するカーネル）と依存関係があるため，待ち合わせカーネルを用いて，境界部分のデータの転送と合流する．CPU1 は，カーネルの投入のみを行い Wilson-Dirac 演算子の完了を待たずに次の処理に移ることで，Wilson-Dirac 演算子の実行中に次の処理の CUDA オーバーヘッドを重ね合わせる．

4.3 CG 法を用いた線型方程式ソルバーの実装

Wilson-Dirac 演算子のみについて CUDA オーバーヘッドを隠蔽する手法を実装しても，それほど大きな効果を得ることは期待できないが，より実践的に Wilson-Dirac 演算子を線型方程式ソルバー内で利用する場合を考えると，より大きな効果が得られることが期待できる．本研究では，BiCGStab 法[8]を用いた反復解法ソルバーを用いて，ソルバー全体に，提案手法による CUDA オーバーヘッドの隠蔽手法を実装した．

BiCGStab 法は，次に示す擬似コードのような反復計算を行う．ここで大文字はスピノルを表し，小文字はスカラー値を表す．D()は，Wilson-Dirac 演算子であるが，本研究では収束性を高めるために，Even-odd 前処理を行っているため，実際には D()の部分は，偶数および奇数格子点それぞれについて Wilson-Dirac 演算子を呼び出しているため 2 回続けて処理を行っている．

```
OUT = R = S = IN
rhop = alpp = omgp = 1.0
P = D(S)
RH = R = R - P
P = V = 0.0
do until rr is not small enough
  rho = dot(R,RH)
  bet = rho*alpp / (rhop*omgp)
  P = bet*P - omgp*bet*V + R
  V = D(P)
  aden = dot(V,RH)
  alp = rho / aden
  S = -alp*V + R
  T = D(S)
  omgn = dot(T,S)
  omgd = dot(T,T)
  omg = omgn/omgd
  OUT = OUT + omg*S + alp*P
  R = -omg*T + S
  rr = dot(R)
  rhop = rho
  alpp = alp
  omgp = omg
enddo
```

また, dot()はスピノル配列の内積を表す. 内積計算では, GPU で計算した内積の値を CPU に集め MPI_Allreduce によって全プロセスで合計を求めて GPU に返す必要がある. そのため, この部分についても本提案手法を用いている. それ以外の線形代数はデータ依存があるのみであるので, 同一 CUDA ストリームを用いて逐次計算すれば良いので, 順番に CUDA ストリームのキューに積んでいけば良いが, dot()の結果のスカラー値を参照する場合は図 12 のように, 線形代数のカーネルの手前に待ち合わせのためのカーネルを入れ, CPU2 でスカラー値の計算を行った後に, スカラー値を GPU に転送し続いて待ち合わせカーネルを停止させるカーネルを呼び出す.

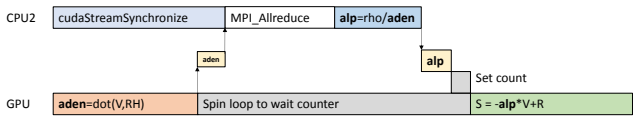

図 12 提案手法による内積計算とその結果を使用するカーネルの間の同期を取る実装

4.4 性能評価

格子 QCD における BiCGStab ソルバーを用いた性能評価に，表 1 に示す OpenPOWER[9]クラスターを利用し，最大で 16 ノードを用いて，Strong Scaling を測定した.

表 1 性能評価に用いた OpenPOWER クラスター

	IBM Power System S822LC [10]
GPU	2*Tesla K80
CPU	2*POWER8
# of cores	2*10
CPU peak	233.6 GFlops / cpu
Memory bandwidth	115 GB/s
Network	Infiniband EDR

また，NVIDIA CUDA Toolkit version 7.5 を使用し，GPU は ECC ありの状態で実行した．また，使用したコンパイラーは, IBM XL C/C++ for Linux V13.1.3 で, MPI 並列化には, IBM Parallel Environment for Linux on Power V2.3 を使用した.

Tesla K80 は GPU カードあたり 2 つの GPU デバイスが搭載されているのでノードあたり 4 つの GPU デバイスがある．ここでは，GPU デバイスあたり 1 つの MPI タスクを用いて実行した．また，提案手法を実現するために，主スレッドの他に MPI の処理のためのスレッドを用意するが，ここでは Wilson-Dirac 演算子の MPI 処理用と，内積演算のためのスレッドの 2 つのスレッドを追加し，MPI タスクあたり 3 スレッドを用いて実行した．スレッド並列化には OpenMP を利用した.

図 13 および図 14 に，それぞれ異なる格子サイズについて 16 ノードまでの実効性能の Strong Scaling 測定値の比較を示す．特に図 13 の 32x32x32x64 の格子サイズでは，従来は 8 ノード以降性能が落ち込んでいたが，提案手法を用いることで，16 ノードでもスケールするようになり，性能が大幅に改善できたことが分かる. 図 14 の 48x48x48x96 の格子サイズについても，大きくスケーラビリティが改善ができていることがわかる.

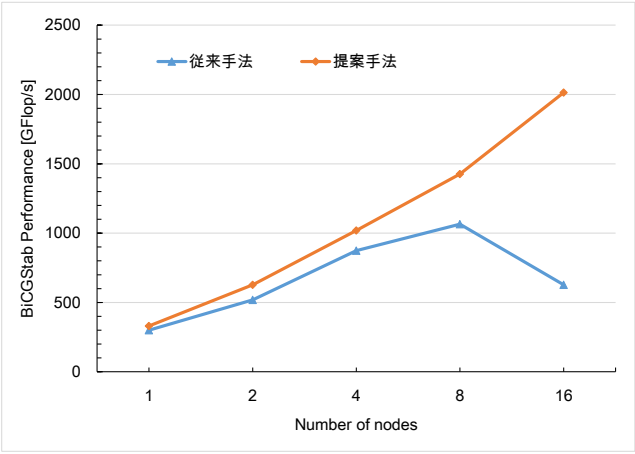

図 13 OpenPOWER クラスターにおける 32x32x32x64 の格子サイズを用いた BiCGStab ソルバーの実効性能の比較

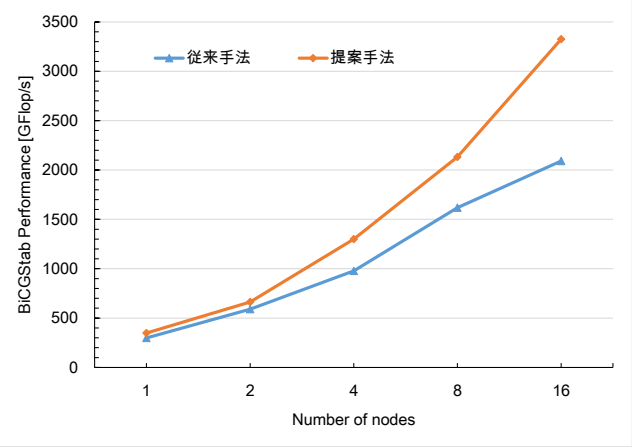

図 14　OpenPOWER クラスターにおける 48x48x48x96 の格子サイズを用いた BiCGStab ソルバーの実効性能の比較

5. おわりに

本研究では CUDA オーバーヘッドが特に Strong Scaling 性能に与える影響に着目し，オーバーヘッドを隠蔽する手法について考察した．CUDA ストリームを用いた非同期処理について，CPU-GPU 間の同期を無くすことで，CUDA オーバーヘッドを GPU の処理に重ね合わせて隠蔽する手法を提案した．CPU-GPU 間の同期を無くすために，CUDA ストリームと CPU での処理を待ち合わせるための小さなカーネルを導入し，CPU 側に追加のスレッドを用いることで，主スレッドをブロックすることなく処理を続けられるようになった．この手法を格子 QCD の Wilson-Dirac 演算子と BiCGStab ソルバーに実装を行い，OpenPOWER クラスター上で Strong Scaling について測定を行った．その結果，本手法を使用しない場合に比べてスケーラビリティが大幅に改善できたことが確認できた．

提案手法による CUDA オーバーヘッドの隠蔽は，Strong Scaling 性能の改善のみならず，将来的により高速な GPU や，NVLink のようなより高速な CPU-GPU 間インターコネクトが登場した際にも，相対的に増大するオーバーヘッドを低減するために役に立つ手法である．

今後は，このような将来登場する新しいシステムにおける提案手法の評価を継続して行っていきたい．また，格子 QCD のみならず他のアプリケーションにも適用できるように，またこの実装をより簡単にプログラミングが可能になるような仕組みを用意したい．

参考文献

[1] 開発者向けの CUDA 並列コンピューティングプラットフォーム, http://www.nvidia.co.jp/object/cuda-parallel-computing-platform-jp.html
[2] NVIDIA Visual Profiler, https://developer.nvidia.com/nvidia-visual-profiler
[3] NVIDIA NVLINK 高速インターコネクト, http://www.nvidia.co.jp/object/nvlink-jp.html
[4] Justin Luitjens, CUDA STREAMS BEST PRACTICES AND COMMON PITFALLS, GTC2014.
[5] Jiri Kraus, An Introduction to CUDA-Aware MPI, https://devblogs.nvidia.com/parallelforall/introduction-cuda-aware-mpi/
[6] JLQCD collaboration: H. Fukaya, S. Aoki, T.W. Chiu, S. Hashimoto, T. Kaneko, H. Matsufuru, J. Noaki, K. Ogawa, M. Okamoto, T. Onogi, N. Yamada, Two-flavor lattice QCD simulation in the epsilon-regime with exact chiral symmetry, Physical Review Letters 98, 172001, 2007.
[7] N. Ishii, S. Aoki, T. Hatsuda, Nuclear force from lattice QCD, Physical Review Letters, July 13, 2007.
[8] H. A. van der Vorst, Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems, SIAM Journal on Scientific and Statistical Computing, vol. 13, no. 2, pp. 631-644, 1992.
[9] OpenPOWER foundation, http://openpowerfoundation.org/
[10] A. Caldeira, M. E. Kahle, G. Saverimuthu, K. C. Vearner, IBM Power Systems S822LC Technical Overview and Introduction, An IBM Redpaper publication.