B-007

DS-CUDA による P2P 機能の評価

Performance Evaluation with Peer to Peer Function of DS-CUDA for Fluid Dynamics Simulation with multiple GPUs

伊藤 一輝† 成見 哲†
Kazuki Ito Tetsu Narumi

1.はじめに

近年、GPU の演算性能は飛躍的な向上を見せており並列演算処理性能が高いことから、GPU を従来のグラフィック処理以外に数値流体計算や Deep Learning などの分野に応用する GPGPU が広く普及してきた。単体 GPU では出来ないくらいの大規模データを扱いたい場合や性能が要求される場合、GPU 間あるいはノード間でのデータ相互通信やメモリアクセスの効率化が必要になる。

DS-CUDA[1][2]はネットワーク上にある GPU を透過的に扱えるミドルウェアであり、MPI (Message Passing Interface)を使わなくても複数ノード上の GPU をあたかもクライアント上に搭載されているかのように使うことが出来る。しかしクライアント-サーバ間の通信時間がボトルネックになり易い。

本研究では、数値流体シミュレーションを複数 GPU で計算する場合に、DS-CUDA に搭載されているサーバ間の P2P (Peer to Peer)機能を用いることで、クライアント-サーバ間の通信を減らし計算効率を向上させることを目標とする。

2.複数 GPU 間通信に関連する技術

2.1 MPI_Alltoallv

現状の GPU では異なるノード間では GPU 間通信が行えないため、一旦ホストメモリ上に通信データを置いておいて MPI を使って間接的に通信を行うことが多い。

MPI の最も基本的な使い方では、送信したいノードで MPI_Send()を呼び、受信したいノードで MPI_Recv()を呼ぶことで一対一の送受信を行う。しかし多くのノードで同時に通信が行われるときは、通信のレイテンシを削減するために、MPI_Alltoallv()を使うことで一回の MPI API 呼び出しで全ての組み合わせのノード間通信を行うことが出来る。

2.2 GPUDirect

GPU Direct[3]は NVIDIA 社が 2010 年 6 月に導入した異なる GPU 間のデータ転送を高速化させる機能である。ノード内の GPU 間転送(P2P)機能により、cudaMemcpy(...,DeviceToDevice) API を使用することによって異なる GPU 間で直接転送出来る。また、異なるノードの GPU 間通信に関しては MPI を経由して通信しなければならない点は変わらないが、InfiniBand ネットワーク使用時に RDMA 機能があり、CUDA-Aware MPI[4]を導入すれば高速な通信が可能になる。

† 電気通信大学院 情報理工学研究科 情報・通信工学専攻

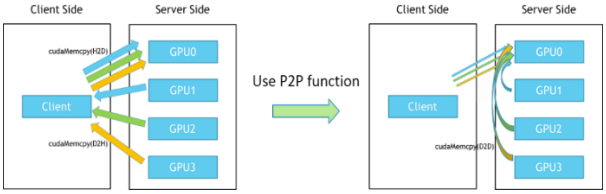

図 1.P2P GPU 間通信の概略図

3.評価実験

3.1 システムの概要

DS-CUDA システムは、ユーザがアプリケーションを実行するクライアントと GPU を搭載した複数のサーバで構成される。お互いは InfiniBand ネットワークで接続されている。ユーザの CUDA プログラムは nvcc ではなく DS-CUDA コンパイラ(dscudacpp)でコンパイルする。サーバ側では DS-CUDA サーバプログラムを常時稼働させておく。クライアントで呼び出された CUDA API の引数はサーバ側に転送され、本当の API はサーバ側の GPU で実行される。

GPU1～3 のデータを GPU0 に転送したい場合を考える。P2P 機能を使用しない場合（図 1 左）は、一旦クライアントにデータを転送してから GPU0 に送るので、クライアント-サーバ間の転送量が多くなる。P2P を使用する場合（図 1 右）は、データ転送自体はサーバ間で行われるため、クライアント-サーバ間転送量を最小限に抑えることが出来る。

3.2 dscudaMemcopies() API

DS-CUDA の場合は、異なるノード間であっても cudaMemcpy(..., DeviceToDevice) API を使う事で見かけ上 GPU 間通信を行うことが出来る。しかし大量の GPU 間通信を行う場合は通信レイテンシが問題となるため、複数の GPU 間通信をまとめた以下の dscudaMemcopies() API を提供している。

```
void dscudaMemcopies(void **dbuf, void **sbuf,
int *count, int ncopies)
```

dbuf[i] : i 番目転送先の GPU メモリのアドレス
sbuf[i] : i 番目転送元の GPU メモリのアドレス
count[i] : i 番目データの送信量
ncopies : データ転送数

転送元および転送先がどの GPU かは、アドレス(UVA)から自動的に判定される。同時実行可能なデータ転送は複数のスレッド上で並列に実行されるため、通信レイテンシをある程度隠ぺいすることが出来る。DS-CUDA を用いた場合は通信レイテンシがボトルネックになり易いことが分かっている[1]。

3.4 アプリケーションプログラム

P2P API の性能評価を行うアプリケーションとして、レーリーテーラー不安定性の流体問題を選んだ[5]。単体 GPU 向けに最適化が行われているコードを、複数 GPU に対応するように修正した。最初に CUDA の native 機能だけを使ってノード内の複数 GPU に対応し、その後 DS-CUDA の P2P API を使って書き変えた。

4 性能評価

データサイズを変更しながら表 1 の検証環境を用いて検証した。サーバ側は 1 ノードにつき 1 GPU が搭載されている。データサイズは単体 GPU で検証できる最大サイズの(128,128,256)グリッドまでとし、CUDA をそのまま使った(Native)時、DS-CUDA 使用時、DS-CUDA(P2P)使用時で使用台数を変えながら 1000 ステップの計算を 10 回計測して平均した。計算速度(Gflops)と 1 ステップあたりの計算時間(msec)は表 2 のとおりである。

各条件における処理性能を考察すると、データサイズが小さく GPU 数が増えるほど各 GPU で一度に行う処理負荷が減少し、GPU 間転送の時間が相対的に大きくなり性能は低下する。P2P を使わない場合のデータの転送時間をT_{com}、P2P 機能によるデータの転送時間をT'_{com}とすると、下記のように表すことが出来る。

$$T_{com} = 2(N-1) \times p \times T_{step} \times t$$
$$T'_{com} = 2(N-1) \times T_{step} \times t'$$

ここで p はグリッド辺りの変数、Nは GPU の台数、T_{step} はステップ数、t , t' は API 1 回あたりの通信時間である。表 3 は最大グリッド数で 2 GPU 時の 1 ステップ辺りの転送時間(T_{com}/T_{step}, T'_{com}/T_{step})である。

単一 GPU を使用した場合、Native に比べて DS-CUDA 使用時はそれ程性能低下がない。複数 GPU 使用時は性能低下が見られるが、P2P 機能使用時の方が性能低下が激しい。この原因を調査したところ dscudaMemcopies()のレイテンシの影響が大きいと考えられる。表 3 より、データ転送の最小値は DS-CUDA 使用時と比較し DS-CUDA(P2P)機能使用時の方が小さいが、最大値では何十倍も大きくなっている。

表 1.検証環境

OS: Ubuntu 14.04.3 LTS / Fedora 14
CPU: Intel Core i7 920 2.67GHz
Memory: 8GB
GPU: GeForce GTX 780
Compiler:dscudapkg2.4.0 and CUDAToolkit 6.0

表 2.性能評価結果

グリッド数	(32,32,64)		(64,64,128)		(128,128,256)	
速度｜1ステップの時間	Gflops	msec	Gflops	msec	Gflops	msec
Native 1GPU	280.1	0.28	792.8	0.78	972.3	5.08
Native 2GPU	140.5	0.54	655.0	0.94	1272.5	3.88
DS-CUDA 1GPU	79.1	0.98	435.6	1.42	899.2	5.50
DS-CUDA 2GPU	36.7	2.10	241.1	2.57	757.6	6.52
DS-CUDA 4GPU	14.7	6.52	120.7	5.12	448.4	11.00
DS-CUDA(P2P) 2GPU	16.7	4.63	118.9	5.24	593.1	8.35
DS-CUDA(P2P) 4GPU	4.2	19.02	32.1	22.13	299.6	16.55

表 3.1 ステップ辺りの転送時間[最小値(左)と最大値(右)]

グリッド数	(32,32,64)	(64,64,128)	(128,128,256)
T_{com}/T_{step}(Native)	0.25ms-0.58ms	0.45ms-0.94ms	1.26ms-1.63ms
T_{com}/T_{step}(DS-CUDA)	0.51ms-1.14ms	0.90ms-1.53ms	2.45ms-3.26ms
T'_{com}/T_{step}(DS-CUDA P2P)	0.47ms-68.85ms	0.69ms-78.46ms	1.58ms-106.10ms

5 今後の課題について

本研究では流体シミュレーションコードを複数 GPU 向けに修正し、DS-CUDA の P2P 機能を用いた場合の性能評価を行った。dscudaMemcopies API が想定以上にレイテンシが大きいために性能が発揮できていないことが分かった。今後はこの API のどこに時間がかかっているか調査する必要がある。

また、複数 GPU 使用時により大きなグリッド数を扱えるように修正したり、より多い GPU 台数を扱えるように修正する必要がある。また、計算領域のデータサイズや変数の数に応じたメモリ使用量をクライアント-サーバ間で判別しながら最適な GPU 数を求める機能も検討する必要がある。

現状、DS-CUDA はクライアント側とサーバ側の環境設定をユーザが行う必要があり、使用台数や通信プロトコルによる切り替えをユーザ本人がしなければならないため負担が大きい。この負担を軽減するようなツールの開発も期待される。

謝辞

レーリーテーラー不安定性の流体シミュレーションコード(GPU 用に最適化済)を提供頂いた東京工業大学青木尊之教授に感謝致します。

参考文献

[1] 老川 稔, 野村 昴太郎, 泰岡 顕治, 成見 哲, “1,024GPU を使用したレプリカ交換分子動力学シミュレーションの並列化”, 情報処理学会 ACS 論文誌, 7/ 4, pp. 1-14, 2014/12

[2] 成見 哲, 老川 稔, Edgar Josafat Martinez-Noriega, 泰岡 顕治, “マイグレーション機能を備えた GPU 仮想化ツール DS-CUDA”, 情報処理学会研究会報告, 2016-HPC-153/17, 2016/3/2

[3] “NVIDIA GPUDirect”, https://developer.nvidia.com/gpudirect

[4] “An Introduction to CUDA-Aware MPI”, https://devblogs.nvidia.com/parallelforall/introduction-cuda-aware-mpi/

[5]青木研究室(東工大)Research: IDO Scheme (Non-conservative Form), http://www.sim.gsic.titech.ac.jp/Japanese/Research/ido.html