GPU コンピューティングにおける各種オーバヘッド評価

システム情報工学研究科 コンピュータサイエンス専攻 1 年
201120742 藤田 典久
指導教員 朴 泰祐
2011 年 12 月 15 日

1 はじめに

従来は 3D グラフィックスを描画するための装置としてのみしか利用されていなかった GPU に，汎用的な計算をさせる General Purpose computing on GPU (GPGPU) が脚光を浴びている．GPU は，CPU と比較して高い並列演算性能とメモリバンド幅を持ち，NVIDIA 社の Tesla M2090 では，倍精度演算性能で 665 GFLOPS，メモリバンド幅で 177GB/sec に達する．

GPU 単体では，プログラムの実行やデータ転送といった動作することはできない．CPU と GPU 間は PCI Express によって接続されており，CPU からの命令やデータ転送は PCI Express を通じて行われる．PCI Express Gen2 16 レーンの帯域は上り 8GB/sec，下り 8GB/sec の全二重通信であり，CPU のメモリ帯域や，GPU のメモリ帯域と比較して細いため，ボトルネックとなりやすい．また，CPU から GPU への命令も PCI Express を通じて送られるため，GPU の操作はオーバーヘッドを伴う．

近年の GPGPU の普及と，1 台のマシンが接続できる PCI Express のレーン数の増加に伴い，1 台のマシンに 3 台や 4 台の GPU を搭載するシステムも登場しているため，効率的なメモリ転送の戦略や，GPU の制御方法が重要視されている [1, 2]．また，計算を全て GPU に任せ，CPU は GPU 制御やノード間通信のみを行う計算モデルだけでなく，GPU が計算を行ないつつ CPU も計算を行う協調計算型のモデルも用いられている．

本研究では，大規模並列 GPU 環境における科学技術計算を視野に入れ，GPU 搭載サーバにおいて性能低下に結びつく様々なオーバヘッドの定量評価を行い，科学技術計算アプリケーション開発の指針を得ることを目的とする．

2 計算機環境

本稿では表 1 に示す計算機を実験に用いる．1 台のマシンに CPU として Xeon E5645 が 2 台，GPU として Tesla M2090 が 2 台搭載され，図 1 のように IO Hub (IOH) を経由して CPU と GPU 間が接続されている．CPU と GPU はそれぞれが独立したメモリを持ち，直接読み書きはできない．CPU-CPU 間と CPU-IOH 間は QuickPass Interconnect (QPI) で接続され，IOH-GPU 間は PCI Express Gen2 16 レーンによって接続されている．QPI の帯域は 12.8GB/sec あるが，PCI Express の帯域は 8GB/sec しかなく，CPU と GPU の間の通信のボトルネックは PCI Express にある．また，CPU1 と CPU2 はメモリ構成が NUMA (Non Uniform Memory Architecture) となっており，GPU から見てメモリ構成が非対称で，遠くのメモリへのアクセスにはペナルティが生じる．

表 1: 計算機環境．

CPU	Intel Xeon E5630 2.53GHz × 2
GPU	NVIDIA Tesla M2050 × 2
Memory	24GB
OS	CentOS 6.0
CPU Compiler	GCC 4.4.4
CUDA Toolkit	4.0

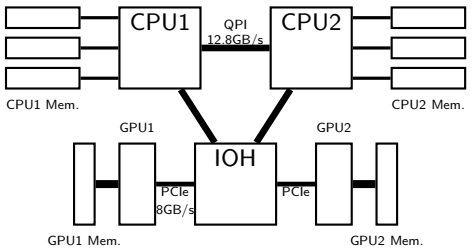

図 1: 計算機のダイアグラム．

2.1 CUDA プログラミング

CUDA プログラミングにおいて，GPU で行う処理は関数単位で記述しカーネルと呼ばれる．また，GPU は CPU 側のメモリを直接アクセスできないため，`cudaMemcpy` といった関数を用いて計算用のデータを転送する．計算用のデータを GPU へ送り，カーネルを起動して計算を行い，結果を GPU から転送す

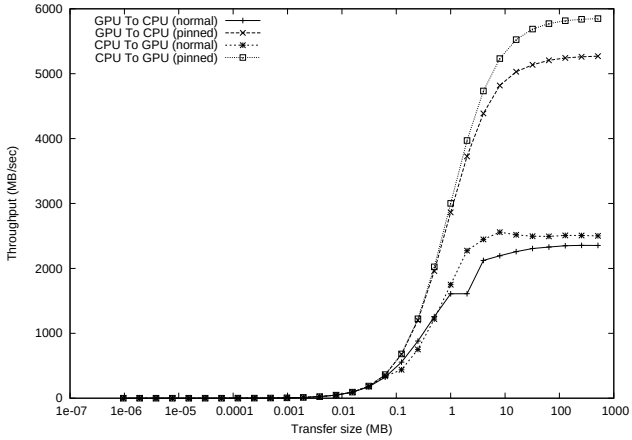

図 2: メモリ転送のスループットの比較.

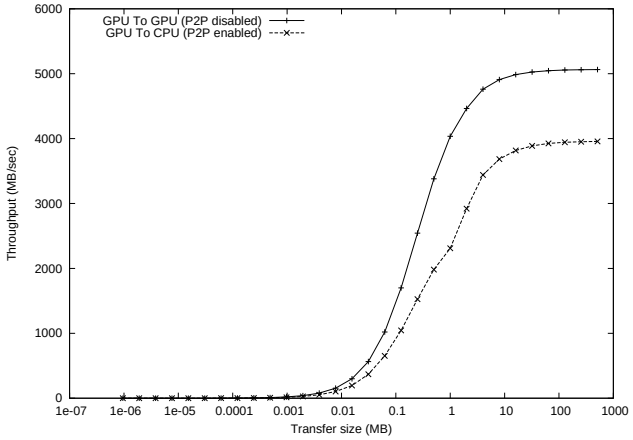

図 3: Peer-to-Peer の有無によるメモリ転送のスループットの比較.

るという手順が基本的な CUDA プログラミングの流れとなる.

3 メモリ転送のオーバーヘッド

3.1 メモリ転送性能

CPU と GPU の間のメモリ転送速度について評価を行う. メモリの転送方向は CPU から GPU への転送と, GPU から CPU への転送について計測する. GPU の転送に用いるための領域の種類は 2 つあり, 1 つは malloc といった通常の方法で確保した領域, もう 1 つは `cudaMallocHost` 関数などによって取得する pinned 領域である. pinned 領域は高速に転送できスワップされないという特徴を持つ. メモリ確保方法による性能差を評価するため, 通常の領域と pinned 領域ついいて各方向に転送速度の測定を行う. ただし, メモリ領域の確保と解放にかかる時間は, 転送時間に含めず, `cudaMemcpy` 関数にかかる時間のみでスループットを計測する.

CPU と GPU 間のメモリ転送のスループットの計測結果を図 2 に示す. CPU から GPU への転送で, pinned メモリの場合は最大 5.84GB/sec, 通常メモリの場合は最大 2.50GB/sec の転送速度が得られ, pinned メモリの方が 2.3 倍高速に転送できることがわかる.

3.2 デバイス間メモリ転送性能

CUDA Toolkit 4.0 より, デバイス間のメモリ転送の API が定義され, 条件が揃えば GPU 同士が CPU を経由せず直接通信できるようになった. GPU 間の直接通信によるメモリアクセスのことを Peer-to-Peer アクセスと呼ぶ.

Peer-to-Peer アクセスの有無を変えた場合のデバイス間メモリ転送速度を図 3 に示す. Peer-to-Peer ありの場合は 5.06GB/sec, なしの場合は 3.96GB/sec であり, Peer-to-Peer ありの方がおよそ 1GB/sec 高速であることがわかる.

3.3 メモリ確保を含めた転送性能

pinned メモリ確保のオーバーヘッドを含めた転送性能を評価するために, 転送に使用する領域を, `cudaMallocHost` もしくは `malloc` 関数で確保し, CPU から GPU への転送を複数回行い, そして確保に用いた関数に対応する解放関数を用いて解放するまでの時間を用いて転送性能を評価する.

転送回数を 1 回と 10 回で計測した性能を図 4 に示す. 1 回限りの転送では, pinned メモリの転送速度を生かしきれず, 通常メモリの方が性能が良い. しかしながら, 10 回転送する場合は, 1MB 未満の領域の転送で同等の性能が, 1MB 以上の領域の転送では pinned の方が性能がよくなることがわかる.

4 NUMA 構成におけるメモリ転送

本稿の実験に用いている計算機は図 1 にあるように, アクセスするメモリによって距離が違う NUMA 構成となっている. GPU とデータをやり取りする場合に, 転送対象のデータの局所性を考慮するべきかどうかを検証する.

実験に際しては, numactl コマンドを用いてプログラムの実行を 1 つの CPU に限定し, cudaSetDevice 関数を用いて操作対象 GPU の選択した. 例えば, 0 番 CPU に実行を束縛し, メモリも 0 番 CPU に接続されているものを使用する場合, 以下のようにコマンドを実行する.

```
$ numactl --cpunodebind=0 --membind=0 \
  command arg1 arg2 arg3
```

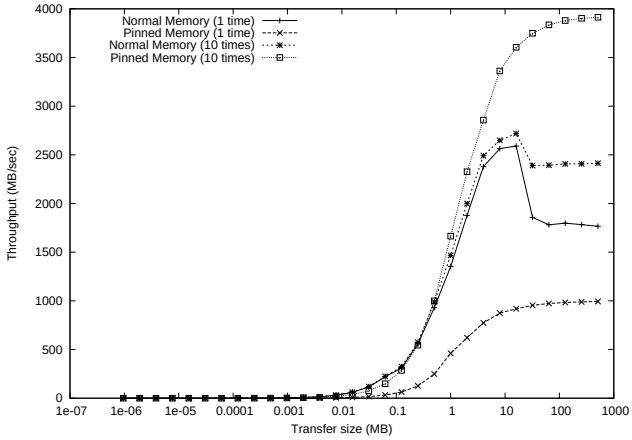

図 4: メモリ確保を含むのスループット比較．ただし，転送方向は CPU から GPU であり，pinned メモリを使用している．

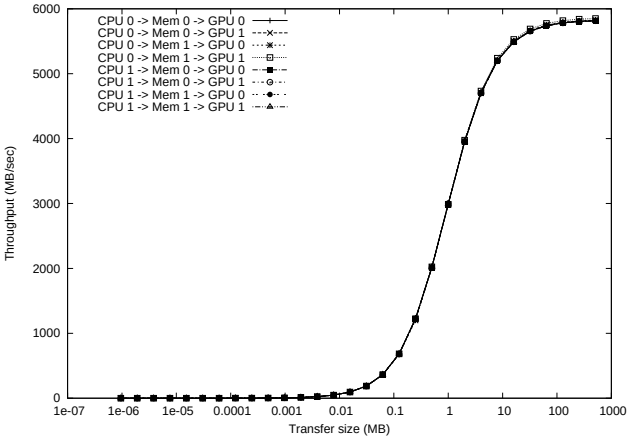

図 5: NUMA を意識したメモリ転送速度の違い．ただし，転送方向は CPU から GPU であり，pinned メモリを使用している．

各 CPU から各 GPU へ pinned メモリ領域を転送した結果を図 5 に示す．転送元の CPU と転送先の GPU はどの組合せで用いても，転送速度に差がないことがわかる．速度差がない理由は，図 1 より実験に用いた計算機では，QPI の帯域が PCI Express の帯域よりも広く，PCI Express の帯域がボトルネックであるためと考えられる．

5 カーネル実行のオーバーヘッド

GPU で実行する処理の単位を決める目安として，カーネル起動のオーバーヘッドを計測する．計測には 2 つの方法を用いる．1 つは CUDA Profiler を用いて時間を計測する方法であり，もう 1 つはカーネルの呼び出しの前後で高精度タイマーを利用して処理時間を計測する方法である．ただし，計測には何も処理をせずに，すぐさま終了するカーネルを使用し，ブロック数とスレッド数はそれぞれ 1 を設定する．

タイマーを用いる方法では，同期の時間をオーバーヘッドに含めないようにするため，カーネルを非同期に呼び出し，前後でタイマーを読み出し，その後，カーネル実行の完了を待機する．タイマーには μs オーダーの精度が必要であるため，x86 系 CPU が内蔵している Time Stamp Counter (TSC) と呼ばれるクロックカウンタを読み出してクロック単位での時間を計測する．

カーネル呼び出しのオーバーヘッドの測定結果は表 2 より $5\mu s$ 以下であることがわかる．ただし，CUDA Profiler によって得られる処理時間の精度は $1\mu s$ しかないのでこの精度での測定である．

また，カーネルの起動した後に `cudaStreamSynchronize` 関数を用いて終了を待機する際の処理時間をタイマーで計測し，カーネルの起動と同期のどちらの方がオーバーヘッドが大きいのかを評価する．ただし，カーネルの起動直後に同期を行うと，カーネルの起動準備といった処理時間を含めて測定してしまう可能性があるため，カーネルを起動した後に 1 秒間待機してから同期を行うこととする．同期にかかる時間は，同期 1 回あたり $162.7\mu s$ かかり，同期オーバーヘッドの方がカーネル起動のオーバーヘッドよりも大きいことがわかる．

表 2: カーネル呼び出しのオーバーヘッドの計測結果．

時刻源	オーバーヘッド
CUDA Profiler	$5\mu s$
TSC	$3.65\mu s$

6 スケジューリングモード

CUDA では，デバイス側の処理の完了を CPU が待つ際の動作モードが Auto, Spin, Yield, BlockingSync の 4 つあり，モードによって CPU の待機時の挙動が大きく異なる．Spin は GPU の待機時にスピンロックを用いて待機する方法であり，最も応答速度が速いが待機している間も CPU がビジー状態のままとなる．Yield は Spin と同様にスピンロックを用いて待機するが，常にスピンをしているのではなく，定期的に処理を他のスレッドに譲る．BlockingSync は CPU スレッドをブロックし，GPU 側の処理の完了を待つ．デフォルトでは Auto が選択されており，Auto は CPU のコア数が GPU の台数以上の場合は Spin，さもなくば Yield と等価である．

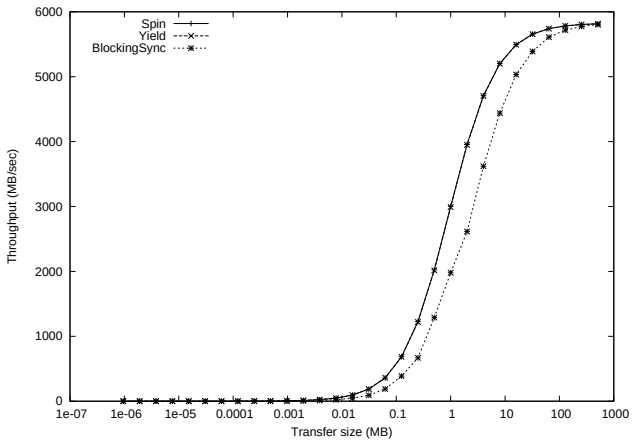

図 6: 各スケジューリングモード時の転送速度．ただし，転送方向は CPU から GPU であり，pinned メモリを使用している．

各スケジューリングモードを使用した際の CPU から GPU へのメモリ転送性能を図 6 に示す．ほとんどの転送サイズで BlockingSync は Spin, Yield よりも性能が悪化している．また，今回の評価では他にスレッドが動作していない状況で測定したため，Spin と Yield の差は見られないが，他に計算や転送を行なっているスレッドが存在する場合では，Spin では GPU を待機しているスレッドが CPU を占有してしまうため，性能が悪化する可能性がある．

7 まとめと今後の課題

データ転送に用いる領域は `cudaMalloc` 関数などを用いて pinned 領域を確保するべきでる．pinned 領域は，CPU から GPU への転送も GPU から CPU への転送も，通常の領域よりも高速であるが，確保に時間がかかるため，複数回転送に利用しなければ，確保と解放に必要な時間を含めた総合的な性能は通常の領域よりも劣る．したがって，確保した領域を使い回すことが重要である．

NUMA 構成を取っているマシンにおいても，どの CPU に属する領域を転送に使用しても，速度は変化しなかった．したがって，アプリケーションを開発する際に，GPU とのデータ転送に用いるメモリ領域の局所性は考慮する必要はない．

BlockingSync スケジューリングモードを使用する場合，Spin や Yield と比較して全体的に性能が低下する．特に小さいメモリ領域の転送など，処理時間が短い処理を行う場合，性能の低下が顕著である．一般的に，CPU と GPU の協調計算を行わない場合は，応答性に優れた Sync や Yield を用いるべきであり，協調計算を行う場合は，CPU で計算している他のスレッドの処理を妨害しないように BlockingSync を用いるべきであるといえるが，最善のスケジューリングモードは，アプリケーションの設計に依存する．

カーネルの起動にかかるオーバーヘッドは小さいため，GPU で実行するカーネルの処理の規模は小さくても構わない．しかしながら，同期のオーバーヘッドは大きいため，同期を取る回数を減らすことが望ましい．例えば，複数の連続していない領域を GPU とやり取りする場合は，同期 API である `cudaMemcpy` を複数回呼び出すのではなく，非同期 API である `cudaMemcpyAsync` を呼び出し，最後に同期を取るべきである．

今後の展望として，本研究で得た知識を用いて生命分野のアプリケーションのひとつである NWChem の GPU 化に取り組んでおり，現在，どの計算を GPU 化すれば高速化に繋るかを検討している [3, 4].

参考文献

[1] TSUBAME2 ハードウェア構成 — TSUBAME 計算サービス. `http://tsubame.gsic.titech.ac.jp/hardware-architecture`

[2] HA-PACS ベースクラスタ — 筑波大学 計算科学研究センター. `http://www.ccs.tsukuba.ac.jp/CCS/research/project/ha-pacs/cluster`

[3] NWChem. `http://www.nwchem-sw.org/index.php/Main_Page`

[4] M. Valiev, E.J. Bylaska, N. Govind, K. Kowalski, T.P. Straatsma, H.J.J. van Dam, D. Wang, J. Nieplocha, E. Apra, T.L. Windus, W.A. de Jong, NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations.