GPGPU によるアクセラレーション環境について

○長屋 貴量

自然科学研究機構 分子科学研究所 技術課 計算科学技術班

概要

GPGPU とは、単純で画一的なデータを一度に大量に処理することに特化したグラフィックカードの演算資源を、画像処理以外の汎用的な目的に応用する技術の一つである。近年、その演算能力は CPU で通常言われるムーアの法則に則った場合とは異なり、飛躍的に向上しており、その演算性能に魅力を感じた各分野での応用が広がってきている。

今回、汎用的な演算サーバの CPU アクセラレータとして採用する場合に必要となる、ソフトウェアの開発環境や移植の容易さ等の特性を知るために、GPGPU 機を構築し、その開発環境の導入を図った。環境導入時にいくつかトラブルに見舞われたが、計算時間についてベンチマークを取ることができた。それらについて、今回報告する。

1 GPGPU について

GPGPU とは、General-Purpose computing on Graphics Processing Units の略であり、単純で画一的なデータを一度に大量に処理することに特化したグラフィックカードの演算資源を、画像処理以外の汎用的な目的に応用する技術の一つである。

近年、CPU の性能向上は、以前のような動作クロック上昇では終焉しており、マルチコア化で性能向上を維持しているが、過去の命令の互換性維持のために、そのトランジスタ数の増加に比例した性能向上を得られなくなっている。またメモリバンド幅も様々な理由で不足している。

これに対し、グラフィックスカードの演算能力は、レガシーに縛られることがないために、CPU で通常言われるムーアの法則に則った場合とは異なり、飛躍的に向上しており、その演算性能に魅力を感じた各分野で応用が広がってきている。

そこで今回、高速分子シミュレーターを運用している当計算センターにおいても、汎用的な演算サーバの CPU アクセラレータとして考慮するために必要な、ソフトウェアの開発環境を導入し、実際に GPGPU の特性やパワーを検討することにした。

2 ベンチマーク

今回 GPGPU マシンを構築するにあたり、以下の計算サーバを構築した。なお、CPU として Intel Core i7 を用いている。

2.1 マシンスペック

＜ハードウェア＞

GPGPU：NVIDIA Tesla S1070-500 (1.44GHz, 30MPU, 240Core, 4GByte Memory) ×4GPU
(Single floating 4.14TFlops, Double floating 345GFlops)

マシン：CPU: Intel Xeon X5550(Nehalem-EP, 4Core, 2.66GHz, L3 cache 8MB) ×2 CPU

Memory: ECC DDR3 1333MHz 24GByte (2GByte x12 [4 channel])

Disk: SATA 500GByte + SATA 1TByte ×5 RAID0

Graphic: NVIDIA Quadro NVS290 (0.92GHz, 2MPU 16Core, 256MByte Memory) ※GPGPU 動作可

Network: GbE x2

Mother board: Intel Workstation Board S5520SC

＜ソフトウェア＞

OS：Linux Fedora 10

GPGPU：CUDA Driver : cudadriver_2.3_linux_64_190.18.run

CUDA Toolkit : cudatoolkit_2.3_linux_64_fedora10.run

CUDA SDK : cudasdk_2.3_linux.run

コンパイラ：Intel Compiler 11.0 + MKL

PGI Compiler 10.0 + ACML (GPGPU サポート)

(OS 以外のソフトウェアは 2009/12 月現在最新のものを用いた)

2.2 ベンチマークの前に

CUDA 関係のソフトウェアを導入後、サンプルプログラムの deviceQuery を実行することで、当マシンに接続している GPGPU のデバイス環境を確認し、当マシン上にある GPGPU ユニット 5 つが全て認識されていることを確認した。しかし、再起動して確認しなおしたところ、デバイスが 1 つも認識されていなかった。これは、管理者権限で GPGPU を用いるプログラムを一度走らせないと、デバイスファイルが作成されないためと考えられた。そこで、起動完了時に一度管理者権限で GPGPU プログラムを走らせるようにした。

また、X Window 起動前では、deviceQuery に表示されるデバイスの表示は、Device0 ~ 4 が Tesla、Device5 がグラフィックボードの順になっていたが、X Window 起動後では、Device0, 2 ~ 5 が Tesla、Device1 がグラフィックボードの順に変わっていた。この理由を把握できなかったため、今回 X Window は起動せず CUI 上で全ての測定を行った。

2.3 ベンチマーク方法

今回、計算速度を比較するために、blas の sgemm/dgemm 関数を用いて、行列の積和計算を行い、その計算にかかった時間を測定することにした。ここで、sgemm は単精度、dgemm は倍精度の行列計算の命令である。また、行列は正方行列であり、行列の一辺の大きさは 2 のべき乗になるようにした。この計算行列の要素の大きさ・次元の大きさ・使う行列の数を考慮すると、CPU 側での計算は一辺が 16384、GPGPU では 8192 が計算可能な次元の最大値となっている。また、あまりに小さな次元の行列では、計算時間が短すぎて正確な時間計測を行っているのか確信が持てなかったので、今回のベンチマークでは行列の一辺の大きさ（次元；要素数とも）が 1024, 2048, 4098, 8192 の 4 通りについて、時間測定の報告を行う。なお、時間測定を 1 回にすると、通常に比べ何らかの割り込みが発生した等で偶然遅い結果が出てしまう可能性を除去できないので、各測定を 500 回繰り返している。

また、当 GPGPU ユニットは、4 つの GPU ユニットが搭載されているため、4 並列で計算が可能と考えられたが、GPGPU を複数個並列で使うには 1 からソースを作成しなければならず、sgemm/dgemm の場合ほど容易に移植できなかった。そのため、今回の発表には間に合わなかった。

また、今回、Intel Compiler の他に PGI Compiler も用いているが、これは当バージョンの PGI Compiler から GPGPU 向けのバイナリーを作成できるようになったためである。しかし今回、うまく作成することができなかったため、こちらも結果を報告することはできなかった。

GPGPU で行列積和計算を行うためのソースコードは、CUDA SDK 中のサンプルプログラム CUBLAS.cu から GPGPU で行列計算する部分を抜き出して作成した。その中で、sgemm/dgemm の命令が出て来る前後の行に gettimeofday 関数を置いて、時間を計測した。また、このソースで、GPGPU 上にメモリ確保する命令や、CPU 側メモリ⇔GPU メモリ間でのデータ転送の命令行を省くことで、Intel Compiler や PGI Compiler でコンパイルできるようにした。なお、GPGPU プログラムでは、GPGPU 向け命令を読み込んだ場合、完了を待たずに次の命令に移る仕様なので（実行速度高速化のため）、全てのスレッドで計算が終わるまで待機する_syncthreads()命令を追加し、計算時間を求めている。

また、当マシンの CPU は 4 Core を 2 つ搭載し、さらに Hyper-Threading を有効にしているので、最大で 16thread の並列計算をオーバーヘッドなく実行できる。そこで、CPU 側で行列計算を行う際には 1～16 threads の並列環境で実行し、その計算時間の差異も調べている。

3　ベンチマーク結果及び考察

3.1　sgemm で計算した場合

Intel Compiler, PGI Compiler, cublas を用いて作成した sgemm の計算時間を図 1 に示す。

左側が Intel Compiler を使った CPU のみでの計算、右側が PGI Compiler を使った CPU のみでの計算、中央が GPGPU での計算時間の結果である。

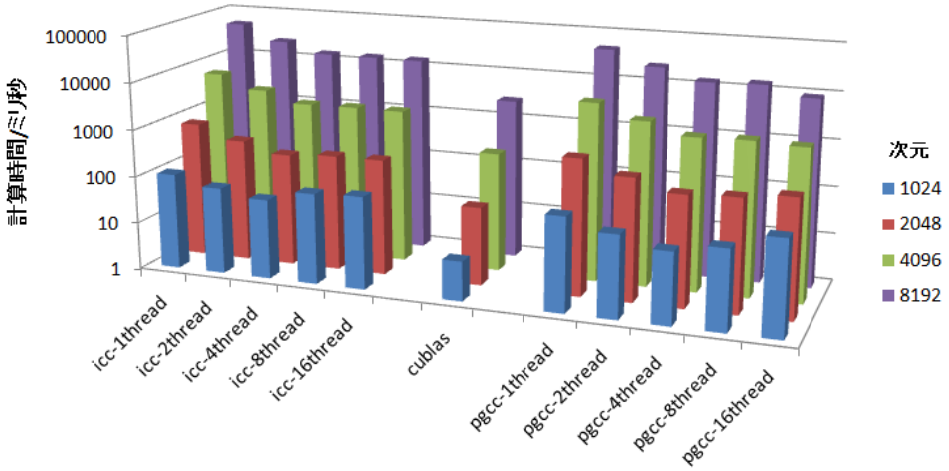

図 1.　sgemm での計算時間プロット

この結果より、中央の GPGPU は、左右両側の CPU での計算より 1 桁程度短い時間で計算完了していることが分かる。これは Intel Compiler 等でさまざまな高速化を試すよりも、GPGPU に計算させるようにするだけで十分所要時間を短くできることを明瞭に示している。

ここで、各条件での比較を容易にするため、GPGPU での計算を基準(= 1)とした場合、各条件では何倍になるかという図を、次に示す。

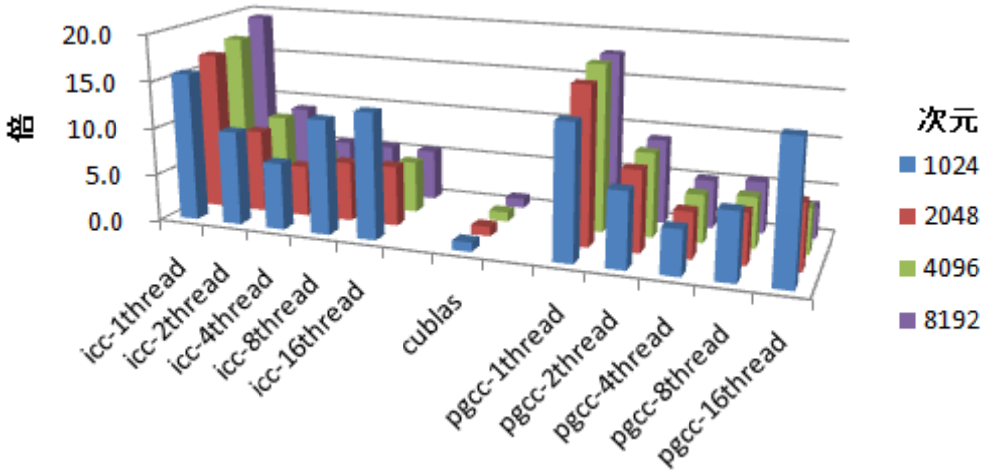

図 2.　GPGPU での計算時間を基準とした場合の計算時間の比(sgemm)

GPGPU での計算時間(中央)を基準にすると、CPU で同じサイズの計算をするために少なくとも 5 倍、最大では 20 倍近くの時間がかかることが見て取れる。

また、Intel Compiler と PGI Compiler を比較すると、若干だが PGI Compiler の方が速い結果となった。

さらに、GPGPU 以外での計算は、次元を落としていっても計算時間が対して落ちないが、GPGPU での計算は、次元と計算時間の間に明瞭な相関関係が見て取れる。

3.2　dgemm の場合

Intel Compiler, PGI Compiler, cublas を用いて作成した dgemm の計算時間を図 3 に示す。

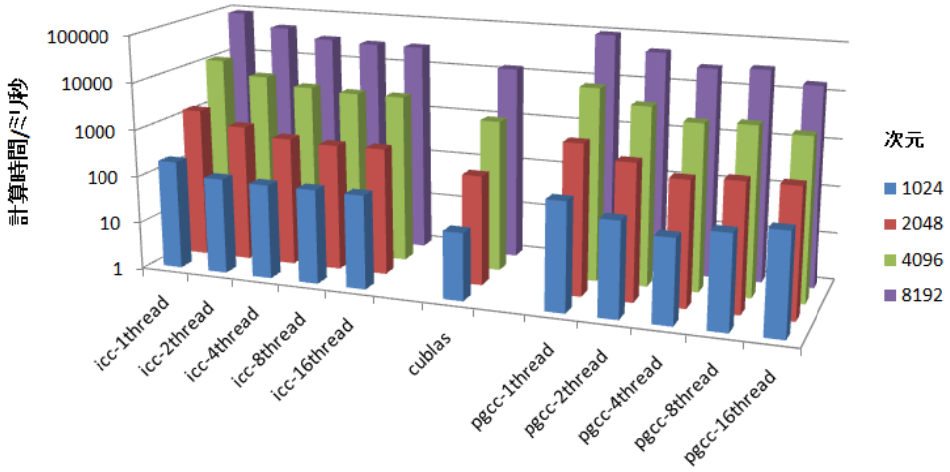

図 3.　dgemm での計算時間プロット

単精度(sgemm)の場合と比べ、CPU との差がやや縮んでいた。行列の各要素のデータの大きさが倍に増えたことを勘案しても、GPGPU の計算時間の増加は CPU での増加に比べかなり大きい。当センターでよく用いられるのは倍精度の方だということを考慮すると、GPGPU にフル対応しても速度の違いが単精度の時ほど大きく感じられないことを予見させる。

また、GPGPU での計算(中央)を基準とした場合、他の方法で何倍の時間かかっているのかを示すプロットを次に示す。

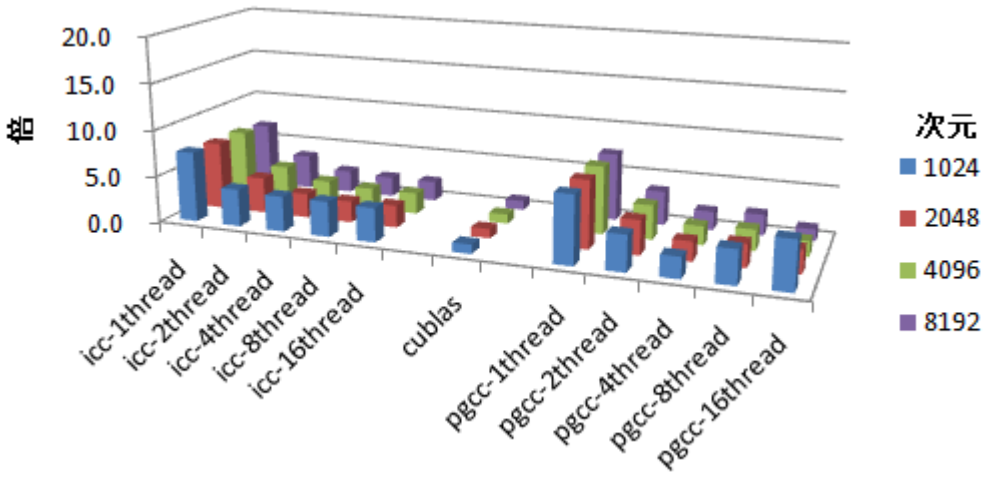

図 4.　GPGPU での計算時間を基準とした場合の計算時間の比(dgemm)

この場合、先の場合ほど顕著ではなく、1.3 ~ 7 倍程度の差が見られる結果となった。GPGPU の倍精度の計算は CPU より多少速いという感触である。

3.3　考察

GPGPU で計算すると単精度で数百倍、倍精度でも 4 倍程度速くなったという話が Web 上で見受けられるが、実際に自分で測定したところ、単精度では 5 ~ 20 倍程度、倍精度では 1.3 ~ 7 倍程度速かった。GPGPU が高速なのは間違いないが、とても数百倍も速くなるとは感じられない。これは、比較対象とした元の CPU が貧弱だったからではないかと思われる。また、倍精度での差は、コンパイラでさまざまな高速化を試すことで、なんとか克服できるレベルであり、期待外れな結果であった。

なお、当マシンはかなり高スペックなもの（Intel Core i7、DDR3 メモリ、GPGPU ユニットと PCI-Express × 16 で接続）を用いているために、GPGPU のアドバンテージが対して目立たなくなった可能性もある。

Intel Compiler に比べ PGI Compiler が、若干速い場合が散見された。これは PGI Compiler の方が後発だからであろう。

スレッド数を増加させることで計算時間が短くなることは予見していたが、スレッド数を大きくするにつれてどれでも速くなるわけではなく、行列の次元に応じて最速なスレッド数が異なるという結果は、予想外であった。スレッド数で分割したことで短くなる計算時間と、分割で発生するオーバーヘッドの時間とのトレードオフの結果だろうと考えている。

CPU 側で行列計算を行わせた場合、次元数を減らしていくにつれて計算時間の減少の割合が小さくなるのに対し、GPGPU 側では、次元数を減らした分だけ計算時間が確実に減少している様子が見て取れた。これは、CPU 側では割り込み処理の発生などで足を取られるのに対し、GPGPU ではそのような割り込みがないために、純粋に次元数に比例した計算時間を示したのだろう。

また、GPGPU ユニットとデータをやり取りする際に生じるオーバーヘッドについて、今回の結果に載せていないが、行列の次元がある程度より小さい（2 桁以下）だと、ある値（約 3 秒）のオーバーヘッドがあり、これより大きいと、その次元の大きさに応じて 4 ~ 6 秒程度のオーバーヘッドが存在しているようである。

4　まとめ

Intel Core i7 と GPGPU で行列計算時間を比較したところ、GPGPU の方が単精度では 5～20 倍、倍精度では 1.3～7 倍程度速いという結果であった。これは、期待していたほど速くはなく、当センターでよく用いられる倍精度のプログラムでは難しいが、通常の単精度のプログラムで、さらに行列計算がボトルネックなものならば、大幅な速度向上が実感できるだろうと言える。

そして、未だに GPGPU の複数ユニットを同時に使用したり、PGI Compiler で GPGPU 向けバイナリーを作成できたりしていないので、こちらも速くテストできるようにし、1 ユニットだけの場合とどれほど差があるのか調査する必要がある。

また、つい先日、倍精度の計算速度を大きく向上させた製品が夏くらいに提供可という話も出たので、そちらにも関心の目を向けているところである。

参考文献

[1]　青木　尊之、額田　彰、”はじめての CUDA プログラミング”、工学社、平成 21 年 11 月、P95-P100

[2]　http://www.nvidia.co.jp/object/cuda_home_jp.html