これからの並列計算のためのGPGPU連載講座(V)
疎行列ベクトル積を題材としたCUDA最適化プログラミング

大島聡史
東京大学情報基盤センター

1 はじめに

第五回の今回は、スーパーコンピュータ上で利用されているアプリケーションにおいても多く利用されている計算の一つである「疎行列ベクトル積」(零要素の多い行列をベクトルにかけた積)を題材として、CUDAを用いたさらなる最適化プログラミングについて解説する。

CUDAを用いた最適化プログラミングの方法については、すでに連載第三回でも紹介した。第三回ではGPUに適した計算を行う行列積(密行列と密行列の積)を題材としたため、効果の大小はあるものの各種の最適化の効果が容易に発揮され、最終的には高い性能を得ることができた。一方で今回対象とする疎行列ベクトル積は間接アドレス参照やランダムアクセスが発生しやすく、データ量に対する計算量も多くないため、行列積と比べて性能向上が容易ではない。事実、第三回の行列積が最大で200GFlops以上の性能を達成できているのに対して、今回の疎行列ベクトル積では多くの問題において数GFlops程度の性能しか達成できていない。しかし、数GFlopsを達成するために行っている最適化は他のアプリケーションでも十分参考になるだろう。

本稿では以下の計算機環境を用いて性能測定を行っている。既により新しいアーキテクチャのGPUが登場しているが、残念ながら最新世代のGPUでは最適化が間に合っていないため今回は使用を見送った。(一般的に、CUDAにおいては多くの旧世代GPU向けのプログラムを新世代GPUでも動作させることができる。しかし性能最適化パラメタについては再検討が必要なことが多い。)

表1 実験環境

CPU	XeonW3520 (4 コア, 2.67GHz)
メインメモリ	DDR3-10600 12GB
GPU	GeForceGTX285 (240 コア, 1.48GHz, 1.0GB, Compute Capability 1.3)
OS	CentOS 5.4 x86_64
開発環境	CUDA Toolkit 3.1, CUDA SDK 3.1

2 対象問題の設定とシンプルな実装の例

疎行列ベクトル積においては対象問題や行列形状の特性にあわせて様々なデータ格納形式が用いられているが、今回は汎用性が高く理解しやすいCRS(Compressed Row Format)形式の入力行列を対象として最適化を行う。CRS形式の行列格納方法およびCRS形式を用いた疎行列ベクトル積のCPU版Cプログラム(OpenMP指示子挿入済み、主計算部分)を図1に示す。疎行列ベクトル積は行単位の計算が独立しているため容易に並列化を行うことができる。またこの

Cプログラムは、Block数およびThread数を1に設定しさえすれば、CUDAプログラムの計算カーネルとしてもこのまま利用可能なプログラムである。

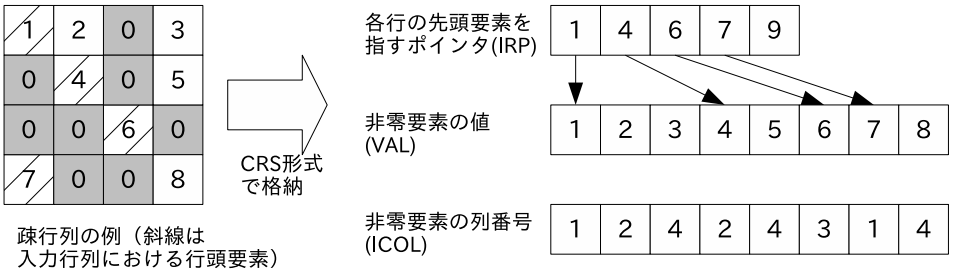

図 1　CRS 形式および CRS 形式を用いた疎行列ベクトル積の C プログラム (OpenMP 指示子挿入済み、主計算部分)

3　シンプルな行単位の並列化実装

3.1　行単位の並列化実装

疎行列ベクトル積は行単位の並列化が非常に容易であり、CPUを用いた並列計算はOpenMPなどを用いれば容易に実装することができる。しかしCUDAにおいて行単位の並列化を行う場合、CUDAにはMPとSPの2階層のプロセッサがあるため、これらのプロセッサにどのように行単位の割り当てるかが課題となる。なおここでは、対象問題のデータサイズ(入力行列の一辺の長さ)が十分に大きい(MP数×SP数より大きい)場合のみを考える。

はじめに各行の計算をGPU全体のSPに割り当てることを考える(図2-A)。この場合はGPUの持つ高い並列性を活用できる一方で、同一MP内の各SPがそれぞれ別の行のメモリをアクセスすることになるため、連続メモリをアクセスできないことが多くなる。そのためコアレスなメモリアクセスとならずに性能が上がらないことが容易に想像できてしまう。たとえば、入力行列が対角行列である場合、単純に各行の計算をSPに割り当てることで連続メモリアクセスとなるため、良い性能が得られることが期待できる。また帯の幅があまり大きくない帯行列であれば、ある程度のメモリアクセスがコアレスなメモリアクセスとして扱われることが期待できるため、やはり良い性能が得られることが期待できる。しかし一般の疎行列を考えた場合には、同一MP内のSPは不連続なメモリをアクセスすることになるうえに、各行の要素数にばらつきがある場合にはSP毎の計算回数が異なることで遊んでしまうSPが増えることになる。このように、この割り当て方法は入力行列の形状が性能に与える影響が大きく、良い性能が得られるか否かは問題形状に依存することになる。

続いて各行の計算をMPに割り当てる場合(図2-B)だが、この場合は行内の計算をどうするかが重要である。各行の計算を各MP上の1SPだけで行うことはできるが、これでは全ての計算を1SPで行った場合と比べてMP数倍程度の性能しか期待できない。より高い性能を得るためには、行内の計算をSPを用いて高速化する必要がある。幸いにもCRS形式における行単位のメモリは連続しており、また疎行列ベクトル積における行単位の計算は入力行列の各要素と入力ベクトルの積を単純に足し合わせたものであるため、SPを用いた並列リダクション演算によりSPを有効活用可能な並列計算が行える。ただし、メモリのアラインメントが揃うか否か

が入力行列の形状に依存するという問題はある。以上からこの割り当て方法は、特に入力行列の各行における非零要素の数が多い場合に、各行の計算をSPに割り当てた場合より良い性能が期待できる。

なお、図2ではSP毎・MP毎に入力行列を1行ずつ割り当てているが、実際には行数が利用可能なSP数やMP数(Thread数やBlock数)を大きく越えることが考えられる。その際にいわゆるブロック割り当てを行うか、サイクリック割り当てを行うかが、連続メモリアクセスの観点から性能に大きな影響を与える可能性があることを考慮する必要がある。

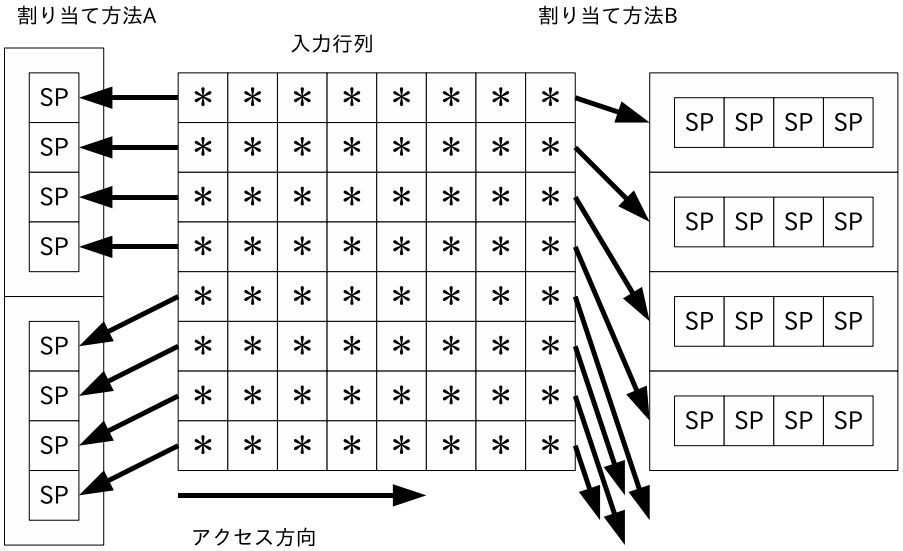

図 2　GPU に対する計算の割り当て方法

以上、ここまでは入力行列の形状に対するMPとSPの割り当てについて見てきたが、入力ベクトルの形状も性能に大きな影響を及ぼす要因となり得る。そもそもCRS形式は入力行列へのアクセスは連続メモリアクセスになる一方で、入力ベクトルへのアクセスは入力行列の形状依存である。たとえば、帯行列のようにベクトルへのアクセスが完全な連続アクセスとなる入力行列形状もあるが、逆にほぼランダムと言えるようなアクセスとなることもある。仮に入力ベクトルが全て高速なSharedMemoryに格納できてしまえばランダムアクセスでも良い性能が得られるが、わずか16KB(Compute Capability 2.0のGPUでは48KB)の容量では大きな問題には対応できない。ConstantMemoryやTextureMemoryを利用することも考えられるが、やはり大きな問題には対応できないため根本的な解決にはならない。任意の入力行列形状に対する入力ベクトルへのランダムアクセスを削減するためには、行列格納形式を変更するという抜本的な対策が必要である。

3.2　性能評価 1

図3および図4に、いくつかの入力行列に対して疎行列ベクトル積の性能を測定した結果を示す。図4は、問題設定と実装によっては図3だけでは性能を視覚的に認識できないケースがあるために、Y軸を対数表示させたものである。実験に用いた入力行列の種類(形状特性)については表2の通りであり、また今回用いた疎行列ベクトル積の実装は以下の通りである。

1. 逐次実行：性能比較用に1Block,1Threadで実行
2. 行単位並列化：各行を各SPに割り当てる[ブロック割り当てとサイクリック割り当てそれぞれを掲載]
3. 行内並列化：各行を各MPに割り当てる(行内の計算は複数SPによる並列リダクション計算)[行単位と行内それぞれについてブロック割り当てとサイクリック割り当てを行い測定、グラフには最も高性能なもののみ掲載]

なおGPUを用いた疎行列ベクトル積の性能測定範囲は、GPUに対して演算開始命令を指示する直前から、GPUによる演算が全て終了するまでとする。入力行列および入力ベクトルのデータをCPU-GPU間で送受信する時間は含めない。

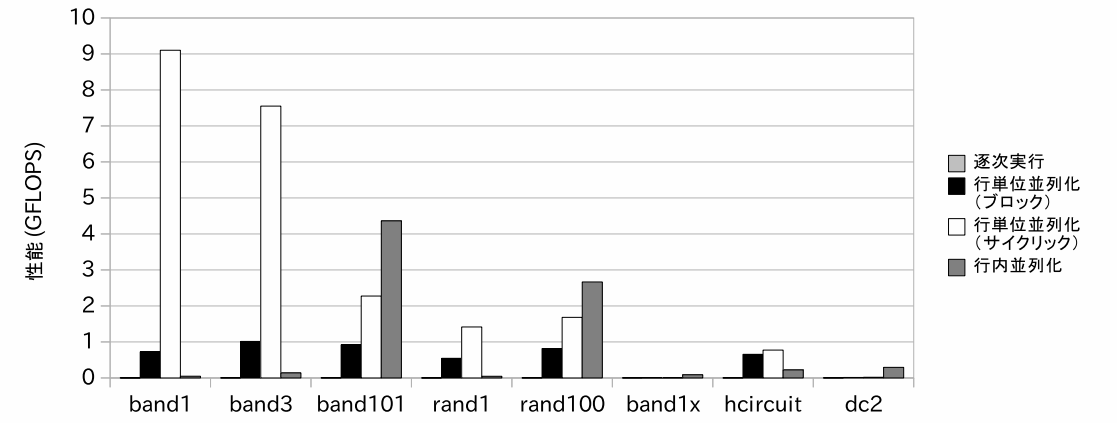

図 3　性能評価 1

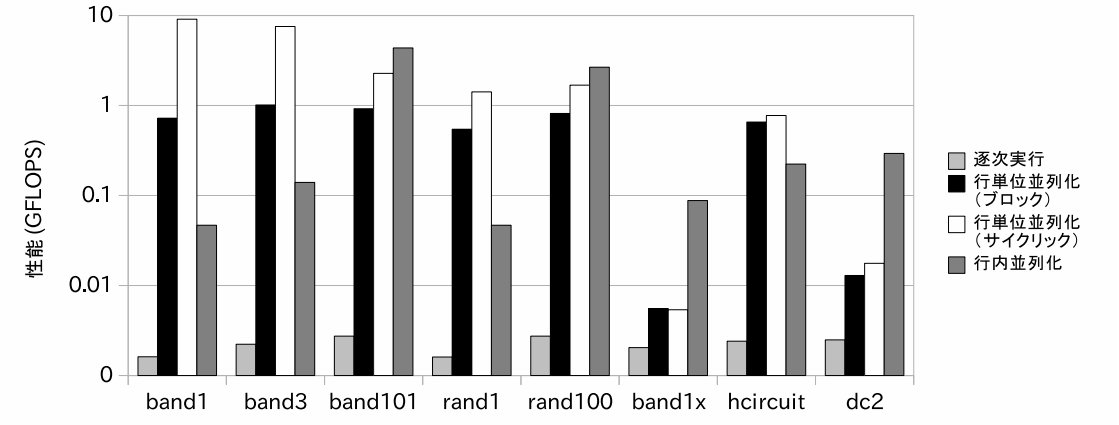

図 4　性能評価 1(対数グラフ版)

実験の結果を見ると、まず逐次実行では入力行列・入力ベクトルの形状に関わらず非常に性能が低いことが確認できる。次に、band1やband3において行単位並列化(サイクリック)が高い性能を得ていることがわかる。これらの問題では行毎の非零要素数が1や3と小さいため、同一MP内のSPが常に近い範囲へとメモリアクセスを行い、コアレスなメモリアクセスとなっていることが考えられる。一方で同様に行単位の非零要素数が少ないrand1の性能が低いのは、

表 2　性能評価に用いた入力行列一覧

行列名	一辺の長さ	全非零要素数	行列形状の特徴
band1	2,000,000	2,000,000	対角行列
band3	2,000,000	5,999,998	三重対角行列
band101	200,000	20,197,450	帯幅 101 の帯行列
rand1	2,000,000	2,000,000	各行に 1 要素、列はランダム
rand100	200,000	20,000,000	各行に 100 要素、列はランダム
band1x	2,000,000	3,999,999	band1 の先頭行のみ全要素を非零要素にしたもの
hcircuit	105,676	513,072	フロリダ行列 [5]、非零要素の配置の偏りが小さい
dc2	116,835	766,396	フロリダ行列、非零要素の配置の偏りが大きい

入力ベクトルへのアクセスがランダムであるコアレスなメモリアクセスとなっていないことが原因であると考えられる。

続いて行単位の非零要素数がやや多いband101やrand100を見てみると、行内並列化が良い性能を発揮している。これも同一MP内のSPによるメモリアクセスがコアレスなメモリアクセスとなったことによるものと考えられる。一方、単純な形状の入力行列であるband1xについてはいずれも非常に低い性能となってしまっている。これは、band1xでは先頭行にのみ非常に多い非零要素があるため、実行時間のほとんどが先頭行の計算に費やされたためである。

以上の入力行列よりも実用的な入力行列であると思われるhcircuitとdc2については、非零要素の配置の偏りが小さなhcircuitでは各種の並列化が若干の性能向上を得ている。また偏りが大きなdc2では行内並列化が若干の性能向上を得ているものの、非常に低い性能であり、非零要素の配置の偏りによる影響は大きいと言える。

以上のように、CRS形式を用いた疎行列ベクトル積のシンプルな並列化だけでも並列化の方法と入力行列の形状次第で大きく性能が変わることがわかる。なお、今回用いた各実装はBlockとThreadの数の調整やループアンローリングなど、詳細なパラメタチューニングの余地が残されていることを付記しておく。

4　SegmentedScan方式による並列化実装

4.1　SegmentedScan 方式

前章の実験結果からわかるように、行単位の並列化により良い性能が得られている問題もいくつかある一方で、band1xやdc2といった入力行列ではいずれの実装でも非常に低い性能となってしまった。これらの問題で全ての実装が著しく低い性能となった主な理由は、行毎の要素数にばらつきがあるため行毎の実行時間にもばらつきが生じ、結果として最も実行時間のかかった行の性能が全体の性能を左右してしまっているためである。SegmentedScan方式はこの問題を解決可能な実装方法の1つである。

SegmentedScanとはBlellochら[1]が提案したScanアルゴリズムの1つであり、これを用いることでベクトル計算機上で疎行列ベクトル積を高速に実行することができる。

SegmentedScan方式による疎行列ベクトル積の概要を図5を用いて以下に示す。CRS方式では入力行列を行頭要素ポインタ(IRP)・非零要素の列番号(ICOL)・非零要素の値(VAL)に分割して保持していたのに対して、SegmentedScan方式では入力行列を固定長のセグメントベクトルに分割して扱い、FLAG行列(行頭要素をTとする行列)とセグメント末尾・行頭配列および非零要素の値に分割して保持する。計算時には各セグメント毎にセグメント末尾から入力行列を走査し、対象要素がセグメント先頭要素または行頭要素ではない場合には前の要素に行列とベクトルの要素の積を引き継ぎ、対象要素がセグメント先頭要素または行頭要素の場合には行列とベクトルの要素の積と引き継がれた積を加算する。その後、セグメント先頭が行頭要素でない場合には前のセグメントにセグメント先頭要素の計算結果を伝搬させることで、疎行列ベクトル積を求めることができる。並列化の際にはセグメントベクトル単位で計算を行うことで、CRS方式の問題であった実行時間のばらつきによる性能低下を防ぐことができる。

さらに櫻井らがSegmentedScan方式を元に実装したBranchlessSegmentedScan方式[2]では、セグメント情報の格納方法や計算順序の変更を行うことで、スカラ型の並列計算機上でも高速に実行できるようになっている。(この実装はXabclib[6]の計算カーネルの1つとして利用されている。) またSegmentedScan方式を用いたGPU向けの疎行列ベクトル積の実装としては、CUDA sdkに同梱されているライブラリCUDPP[3]にて再帰的な実装を施したSegmentedScan[4]が利用されている。SegmentedScan方式にはセグメントベクトル間で計算結果を引き継ぐ処理が必要であり、セグメントベクトルの数が多い場合にはこの引き継ぎ計算の実行時間が全体に占める割合が大きくなってしまう。再帰的なSegmentedScan方式は、この引き継ぎ計算にもSegmentedScan方式を用いたものである。

4.2 インデックスを用いた SegmentedScan 方式

一方、筆者らもCUDA向けに独自の最適化SegmentedScan方式を実装しており、問題設定次第ではCUDPPを超える性能を達成している[7]。以下本節では我々のCUDA向けSegmentedScan方式の内容を解説する。

図6に本手法のデータ保持方法と計算手順の概要を示す。我々のSegmentedScan方式では、事前にセグメント行列を元にして行列形状についてのヒント、すなわちセグメント行列の要素がセグメント先頭要素もしくは行頭要素から何番目の要素であるかを示すインデックスを計算しておく。実行時には各要素においてセグメントベクトル長の半分から始まり計算ステップのたびに半減していく閾値を用意し、各要素のインデックスが閾値以上閾値の2倍未満の場合には閾値分だけ前の要素に値を足すという処理を、閾値が0に到達するまで繰り返す。一見すると閾値とインデックスとの判定などにおいて同一MP内のSP間における分岐処理が必要に思えるが、実際には書き込み先配列番号を四則演算で調整すればSharedMemory内でのダミー読み書きに落とし込むことができる。事前にインデックスというヒントを作成していることと、インデックスを用いてセグメント毎に一括計算を行っていることが主な特徴である。なお、実際にはセグメント間のデータ伝搬には再帰的なSegmentedScan方式を用いることでさらなる性能向上を行っている。

我々のSegmentedScan方式をGPU向けにSegmentedScan方式を実装しているCUDPPと比較すると、我々の方式は事前にヒント(インデックス情報)の生成を行うための実行時間やヒントを

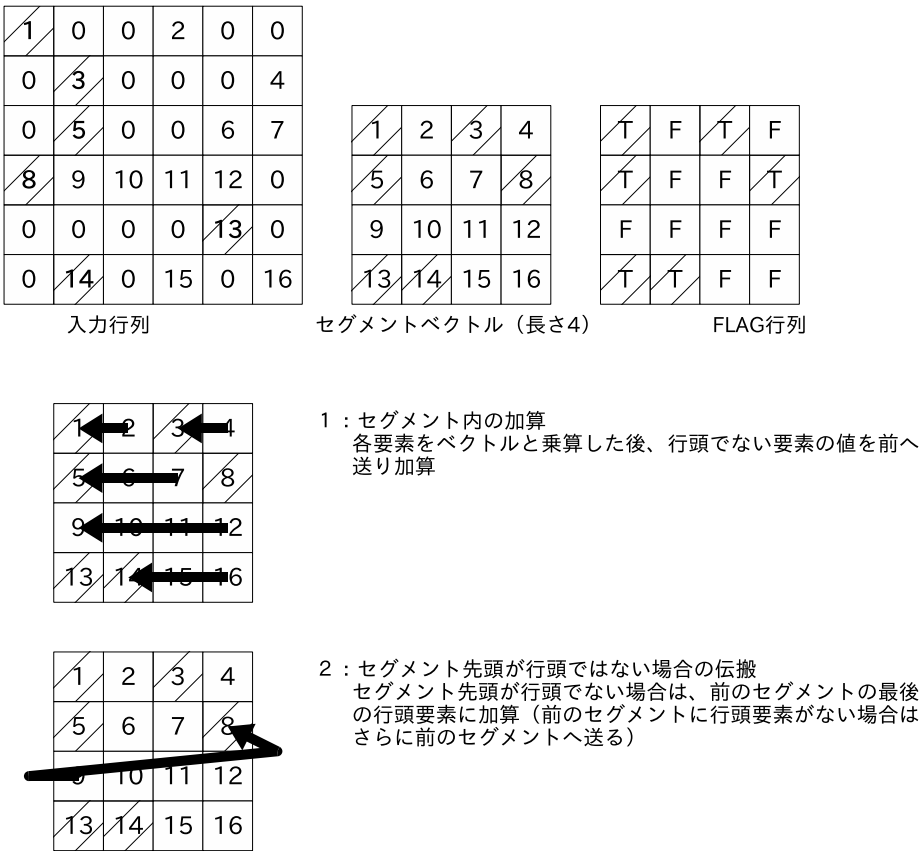

図 5　SegmentedScan 方式のデータ保持方法と計算手順

やりとりするためのCPU-GPU間メモリ使用量増加がデメリットとなる可能性がある。しかし、疎行列ベクトル積は同じ形状の行列に対して複数回計算されることが期待できる。また後述の実験結果からわかるように、SegmentedScan方式が様々な入力行列に対して常に高性能を得られるとは限らないため、入力行列の形状特性を事前に確認してアルゴリズム選択を行うことが重要であると考えられる。入力行列の形状特性を調査する処理と一緒にヒントの生成やメモリ使用量・実行時間などの見積もりを行うことで、デメリットが相殺されることが期待できる。

4.3　性能評価 2

インデックスを用いたSegmentedScan方式とCUDPPの性能を図7および図8に示す。各図には比較用として性能評価1の結果も再掲している。評価に用いた入力行列は性能評価1と同様である。GPUを用いたそれぞれの実装における性能測定範囲も性能評価1と同様に、入力行列および入力ベクトルのデータがGPUに転送された後、GPUに対して演算開始命令を指示する直前から、GPUによる演算が全て終了した後までとする。

実験の結果、インデックスを用いたSegmentedScan方式の性能が入力行列の形状による影響をあまり受けていないことがわかる。入力ベクトルについては、band1とrand1を比較した場合やband101とrand100を比較した場合に明らかなように、特に対策となるような実装をしていないためランダムなものと逐次なものとで有意な性能差が確認できている。同様にCUDPPについても、hcircuitとdc2を見ると入力行列の形状による性能差が小さいが、他の単純な形状の

図 6　インデックスを用いた SegmentedScan 方式

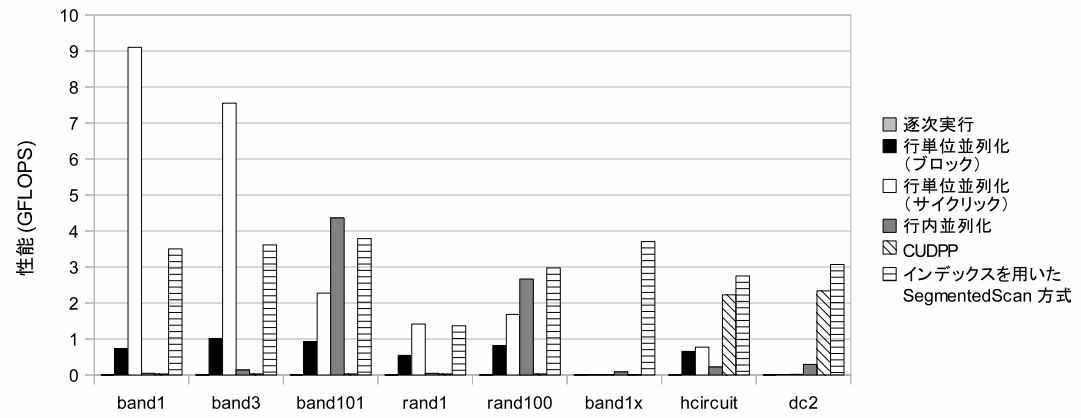

図 7　評価結果 2

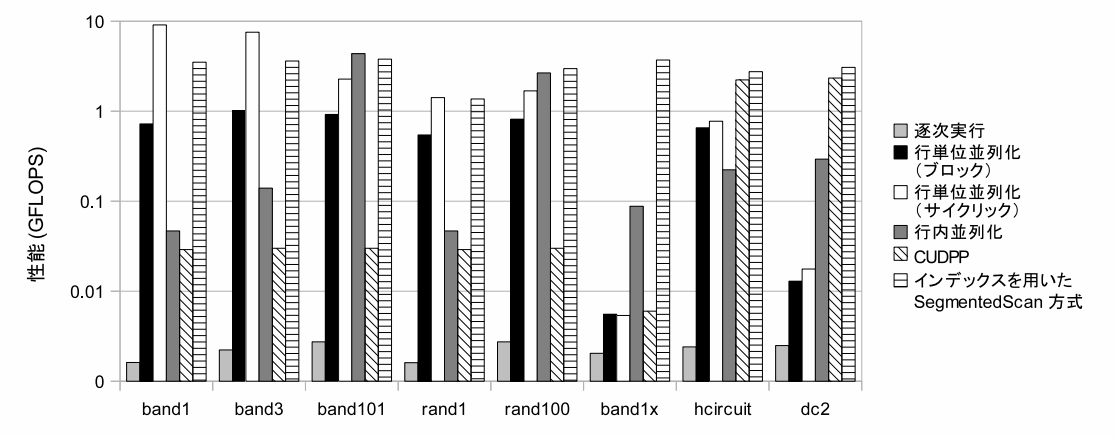

図 8　評価結果 2(対数グラフ版)

入力行列において良い性能が得られていない。これはCUDPPの最適化実装において、入力行列の各行の非零要素数が今回のように非常に小さい場合を想定していないことによるものだと考えられる。また、hcircuitとdc2では、インデックスを用いたSegmentedScan方式がCUDPPを上回る性能を達成していることがわかる。ただし既に述べたように、我々が実装したインデックスを用いたSegmentedScan方式はヒントの生成が必要であるのに対してCUDPPは不要であることから、実際のアプリケーションで利用する場合には常にこの性能評価通りの性能差が出るとは限らない。

ところで、インデックスを用いたSegmentedScan方式の性能を行単位並列化や行内並列化性能評価1と比較してみると、確かに偏りのある行列においては著しく性能が向上している一方で、band1やband3およびband101といった非常に単純な問題では性能を上回れていない。この理由は、特にband1の計算内容を確認してみると明らかなのだが、band1はそもそも各行の非零要素数が1なので、行毎の足し合わせもセグメントベクトル間の足し合わせも不要である。このような問題においてはインデックスを用いたSegmentedScan方式で行っている処理は無駄であると言える。

このことからインデックスを用いたSegmentedScan方式においては、各セグメントベクトル内の要素が全て同じ行だった場合や全て別の行だった場合などにセグメント内の計算をより簡潔にする実装を取り入れれば、多少は性能向上がする可能性がある。しかし、やはり入力行列や入力ベクトルの形状に合わせて最適な計算アルゴリズムを選択することがより重要であると言えるだろう。

以上、第五回の今回は疎行列ベクトル積を題材としてCUDA最適化プログラミングの解説を行った。疎行列ベクトル積はGPUにとっても特に向いている対象問題であるとは言えず、密行列積のように絶対的に高い性能が得られはしない。しかし入力データに合わせたアルゴリズムの選択が重要であるなど、CUDA最適化プログラミングの例としては興味深い題材である。筆者としても今後さらなる性能改善やCRS以外の行列格納形式での最適化実装などを行う予定である。

(次回に続く)

参考文献

[1] Blelloch, G.E., Heroux, M.A. and Zagha, M.: Segmented Operations for Sparse Matrix Computation on Vector Multiprocessors, Technical Report CMU-CS-93-173, School of Computer Science, Carnegie Mellon University (1993).

[2] 櫻井隆雄，直野健，片桐孝洋，中島研吾，黒田久泰，猪貝光祥：自動チューニングインターフェースOpenATLib における疎行列ベクトル積アルゴリズム，情報処理学会研究報告 2010-HPC-125，pp.1—8 (2010).

[3] CUDPP: CUDA Data Parallel Primitives Library, http://code.google.com/p/cudpp/ .

[4] Sengupta, S., Harris, M. and Garland, M.: Efficient Parallel Scan Algorithms for GPUs, Technical Report NVR-2008-003, NVIDIA Corporation (2008).

[5] T.A.Davis: Sparse Matrix collection, http://www.cise.ufl.edu/research/sparse/matrices/ .

[6] Xabclib プロジェクトホームページ, http://www.abc-lib.org/Xabclib/index-j.html .

[7] 大島聡史，櫻井隆雄，片桐孝洋，中島研吾，黒田久泰，直野健，猪貝光祥，伊藤祥司：Segmented Scan法のCUDA向け最適化実装，情報処理学会研究報告2010-HPC-126，pp.1—87(2010).