Vol.2010-HPC-126 No.1
2010/8/3

Segmented Scan 法の CUDA 向け最適化実装

大島聡史[†1] 櫻井隆雄[†2] 片桐孝洋[†1]
中島研吾[†1] 黒田久泰[†3] 直野健[†1]
猪貝光祥[†4] 伊藤祥司[†1]

本稿では Segmented Scan 法を用いた疎行列ベクトル積の CUDA 向け最適化実装について述べる．我々は実装の再利用性に着目した自動チューニングインターフェース OpenATLib の提案を行い，また OpenATLib の提供する機能の一つである疎行列ベクトル積においては Segmented Scan 方式を元にスカラ計算機向けに改良を行った Branchless Segmented Scan 方式を提案している．本稿ではこれらの方式を元にして CUDA 向けの新たな Segmented Scan 方式を考案し実装した．GPU 上で高速実行可能なようにアルゴリズムの改良や各種の最適化を行った結果，偏りの大きな行列に対して NVIDIA GeForceGTX285 上で最大で 3.26GFLOPS の性能を達成した．

Optimized Implementation of Segmented Scan Method for CUDA

Satoshi Ohshima,[†1] Takao Sakurai,[†2]
Takahiro Katagiri,[†1] Kengo Nakajima,[†1]
Hisayasu Kuroda,[†3] Ken Naono,[†1] Mitsuyoshi Igai[†4]
and Shoji Itoh [†1]

We discuss about optimized implementation of sparse matrix vector multiplication for CUDA using Segmented Scan method. We proposed Auto-tuning interface OpenATLib and we also proposed Branchless Segmented Scan method besed on Segmented Scan method for scalar computer as an important new feature of sparse matrix vector multiplication. In this paper, we proposed and implemented new Segmented Scan method for CUDA based on Segmented Scan method and Branchless Segmented Scan method. As a result of optimized implementation, we aimed 3.26GFLOPS on NVIDIA GeForceGTX285.

1. はじめに

高い並列演算性能を持つ GPU は各種の数値計算問題をはじめとして様々な用途に利用されており，GPU を用いた汎用演算 GPGPU(General-Purpose computation using GPUs)[1] の有用性は広く認識されつつある．しかし高性能な GPU プログラムの作成には専門的な知識と技術が必要である．そのため，GPU 向けに最適化された実装方法の研究や，GPU 向けの実装を隠蔽した言語やライブラリへの注目も高まっている．

一方で科学技術計算などに用いられる行列計算ライブラリにおいては，計算対象となるデータの内容や計算環境によってアルゴリズムや最適化パラメタを選択することが性能に大きな影響を及ぼしている．そのため，これらの選択の支援を行う自動チューニング方式が注目されており，我々は実装の再利用性に着目した自動チューニングインターフェース OpenATLib の提案と実装を行っている[2]．

現在我々は CPU 向け数値計算手法と GPU 向け数値計算手法の比較や，比較結果に基づく CPU と GPU の選択機能を備えた数値計算ライブラリの可能性を検討している．その一環として，我々は実行時間に対して入力行列の形状が及ぼす影響が小さい行列ベクトル積アルゴリズムである Segmented Scan 方式[3] および，OpenATLib の実装において櫻井らが Segmented Scan 方式をスカラ型並列計算機環境向けに改良した Branchless Segmented Scan 方式[4] に注目し，CUDA 向けの改良実装を行った．本稿では CUDA 向け実装における最適化手法や性能について述べる．

本稿の構成は以下の通りである．2 章では対象問題である疎行列ベクトル積と既存手法である Segmented Scan 方式および Branchless Segmented Scan 方式について述べ，さらに既存の GPU 向け疎行列ベクトル積についても述べる。4 章では我々が実装した SegmentedScan 法の詳細を述べる．5 章では性能評価を行い，6 章でまとめる．

†1 東京大学 情報基盤センター スーパーコンピューティング研究部門
Supercomputing Research Devision, Information Technology Center, The University of Tokyo
†2 日立製作所 中央研究所
Central Research Laboratory Hitachi, Ltd.
†3 愛媛大学 大学院理工学研究科
Graduate School of Science and Engineering, Ehime University
†4 日立超 LSI システムズ
Hitachi ULSI System Co., Ltd.

© 2010 Information Processing Society of Japan

Vol.2010-HPC-126 No.1
2010/8/3

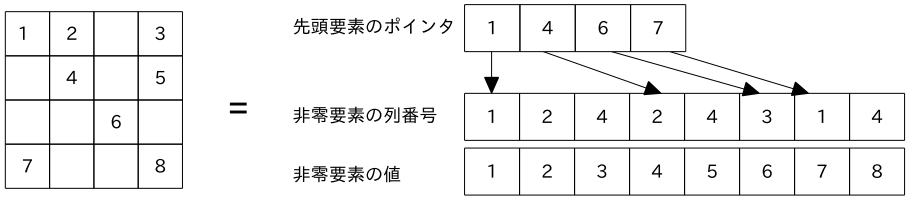

図 1 CRS 形式を用いた疎行列ベクトル積と並列化
Fig. 1 Sparse matrix vector multiplication using CRS format and its parallelization

2. 疎行列ベクトル積と Segmented Scan 方式

本章では，本稿が対象とする疎行列ベクトル積と Segmented Scan 方式および Branchless Segmented Scan 方式について述べる．さらに GPU 向けの疎行列ベクトル積についても述べる．

2.1 Segmented Scan 方式

疎行列ベクトル積は零要素の多い行列 (疎行列) とベクトルによる積を求める計算であり，様々な科学技術計算などのアプリケーションにおいて実行時間の多くを占める計算であるために高速化の要求が大きい．そのため大規模な疎行列を現実的なメモリ容量で扱うために，これまでに様々な行列要素格納方式と計算方法が提案され利用されている．我々は現在，行列要素格納方式として汎用的かつシンプルな CRS(Compressed Row Storage) 形式 (図 1) を用いた行列ベクトル積を対象として，性能最適化やライブラリの作成を行っている．

CRS 形式の入力行列に対する疎行列ベクトル積は，入力行列について行ごとに計算を行うことで容易に並列高速化を行うことができる．しかし，各行の非零要素数にばらつきがある場合には非零要素数が多い行の計算時間が全体の実行時間を律速する．典型的な例として，入力行列のある一行には N 要素，他の行にはそれぞれ 1 要素ずつの非零要素が存在する NxN の正方行列 (以下，本稿中ではこのような行列を “過度に偏った行列” と呼ぶことにする) を入力行列とした疎行列ベクトル積を考える．この積を並列計算する場合，単純な行単位の並列化では N 要素が存在する一行の計算を分割することができないため，並列化しても高い性能を得ることはできない．

一方の Segmented Scan 方式は入力行列を行単位ではなく固定長のセグメントベクトルに分割する．図 2 は CRS 形式で保存されていた疎行列をセグメントベクトルと FLAG 行列に格納し直した例を示している．FLAG 行列は，セグメントベクトルの各要素が入力行列の行頭要素に対応するか否か，を意味するフラグである．これを用いて疎行列ベクトル積を行う場合には，以下の手順で計算を行えばよい．

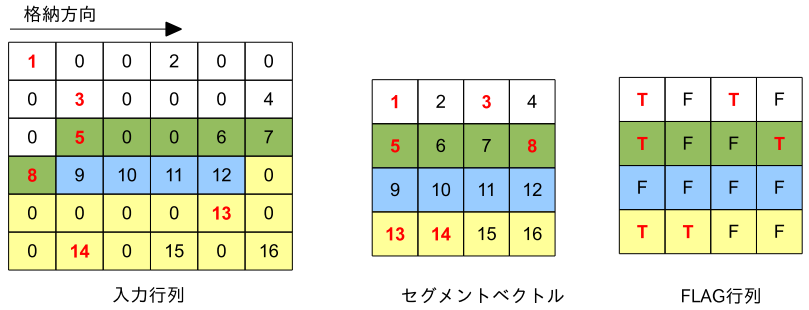

図 2 Segmented Scan 方式におけるデータ格納方法
Fig. 2 Data layout of Segmented Scan Method

(1) 各セグメントベクトルごとに，セグメントベクトルに含まれている部分入力行列ごとに部分行列ベクトル積を行う
(2) 複数セグメントベクトルにまたがった入力行の部分積を統合する
(3) 入力行の先頭要素を抽出する

Segmented Scan 方式では非零要素の配置が性能に及ぼす影響が小さくなるため，過度に偏った行列についても単純な行単位の並列化に比べて非常に高い性能を得ることができる．

オリジナルの Segmented Scan 方式ではフラグを参照しつつ計算を行った．これに対して櫻井らの提案した Branchless Segmented Scan 方式では，直接的にフラグを持つのではなく，入力行列の行頭要素とセグメントベクトルの行頭要素に対するポインタのリストを用意している．これらの変更により計算順序の変更と分岐処理の削減が行われ，スカラ型の並列計算機において高い性能が達成されている．

2.2 GPU を用いた疎行列ベクトル積

GPU はベクトルや行列に対する処理に適したハードウェアであり，古くはグラフィックス処理を用いて GPGPU を行っていた頃から各種の数値計算アルゴリズムの実装が盛んに行われてきた．今日では NVIDIA 社の GPU 上で CUDA[5] を用いて数値計算を行う研究が盛んである．

CUDA を用いた CRS 形式疎行列ベクトル積の並列化については，CPU と同様に行ごとの計算を GPU 上の各プロセッサに行わせることで容易にある程度の性能を得ることができ

© 2010 Information Processing Society of Japan

る．しかしこの計算には入力ベクトルに対するランダムアクセスが必要であり，また入力データに対する参照の連続性や局所性が低い．GPU はランダムメモリアクセスの多い問題やメモリアクセスの局所性・反復性が低い問題に適したハードウェアではないため，本問題は GPU に適した問題とは言い難い．参考として，非零要素の配置が異なるいくつかの行列に対して，CUDA を用いて単純な行単位の並列化実装を行い性能を測定した．その結果，対象問題における非零要素の配置が GPU による演算に適していた場合には 3GFLOPS 程度の性能が得られた一方で，GPU に適さない非零要素の配置である問題については 0.1GFLOPS 未満の極めて低い性能しか得られず，入力行列の形状に大きな影響を受ける結果となった．

一方，CUDA 向けにいくつかの数値アルゴリズムを実装したライブラリ CUDA Data Parallel Primitives(CUDPP)[6] には Shubhabrata[7] らの Segmented Scan 法が取り入れられている．Shubhabrata らの実装では Segmented Scan 方式における “複数セグメントベクトルにまたがった入力部の部分積を統合する” 処理に再帰的に Segmented Scan 方式を利用し，さらに CUDA の命令スケジュール単位 (Warp,CTA) を意識した実装によって高い性能を得ている．Shubhabrata らの実装は我々の実装と想定している利用方法や問題設定が異なるため対等な性能比較をすることは難しいが，4 章にて比較を行うことにする．

3. Segmented Scan 法を用いた行列ベクトル積の CUDA 向け最適化実装

本章では我々の行った最適化実装の内容について述べる．はじめに我々の設定した問題設定について確認した後，実装全体の概要を説明し，特に重要な最適化実装方法の詳細を解説する．

3.1 問題設定

GPU を用いて計算を行うためには CPU から GPU へデータを転送する必要があり，また計算結果を出力するためには GPU から CPU へ計算結果を転送する必要がある．CPU-GPU 間は PCI-Express(x16 Gen2) バスで接続されており，次章の実験環境においては実測値として片方向 6GB/s 程度の速度で CPU-GPU 間データ転送を行うことができる．しかし，CRS 形式を用いた疎行列ベクトル積においては使用するデータ容量に比べて計算量が少ないため，疎行列ベクトル積を 1 回行う時間に対する CPU-GPU 間の通信時間の割合は小さくはない．また，現実のアプリケーションでは疎行列ベクトル積を一度だけ演算することが要求されることは考えにくく，同じ入力行列を用いて何度も繰り返し疎行列ベクトル積を計算することや他の計算と組み合わせて利用することに需要があると考えられる．

以上から，本稿における性能評価の範囲については，CPU から GPU に対して演算開始の指示を送る時点を開始時刻，GPU 上で演算が終了し CPU へ演算結果を書き戻すことが可能となる時点を終了時刻として扱い，計算前後の CPU-GPU 間の通信時間については性能評価の範囲に含めないことにする．計算の途中で CPU-GPU 間の通信を行う必要がある場合には性能評価の範囲に含める．開始時刻においては GPU 上に CRS 形式で格納された入力行列と単純な配列に格納された入力ベクトルが存在し，終了時刻においては入力ベクトルとは別の単純な配列に格納された出力ベクトルが存在する．また，入力行列は書き換えられずに残るものとする．

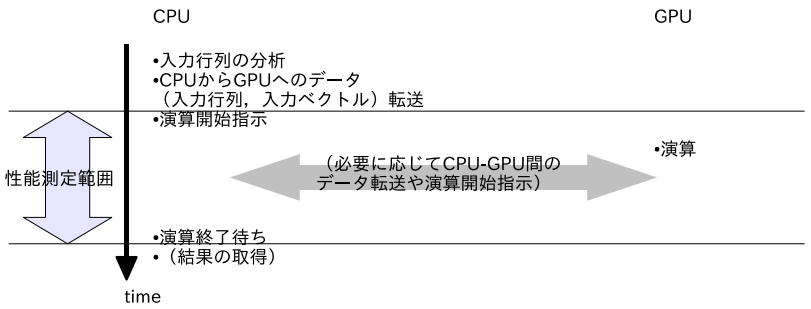

図 3 問題設定
Fig. 3 Problem configulation

一方で，前述のように同じ入力行列を用いて繰り返し計算を行うことを想定した場合，入力行列の形状に関する情報を毎回計算する必要はない．さらに我々は数値計算ライブラリに対する要求として，計算時間を短くすることだけではなく，メモリ使用量を減らすことや計算精度を保証することなど利用者のポリシーにあわせてアルゴリズムの選択をすることが重要であると考えている．この選択においては，入力行列の形状特性などの情報を利用することは重要である．

これらの理由から，本稿では CRS 形式を用いた疎行列ベクトル積を対象としているが，演算を開始する時点で入力行列の形状に関していくつかの追加情報を用意しても良いことにする．追加情報を用意するための準備時間や必要メモリ容量については別途 4 章にて確認する．

図 3 は問題設定をまとめた図である．本稿では以上の問題設定の元，実行時間の短縮を目標として最適化実装を行った．

© 2010 Information Processing Society of Japan

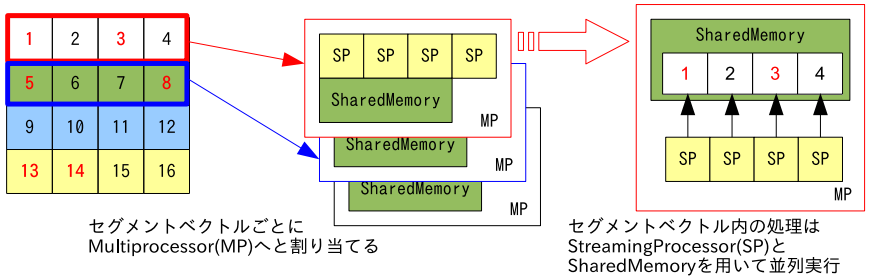

図 4 CUDA と Segmented Scan 法の対応付けの概要
Fig. 4 Outline of assignment of CUDA and Segmented Scan method

3.2 CUDA 向け Segmented Scan 法の基本的な実装

図 4 に我々の実装における CUDA と Segmented Scan 法の対応付けの概要を示す．

CUDA の並列実行モデルは階層性のあるプロセッサとメモリのモデルを有しているため，これを対象問題と適切に対応づけることが重要である．特に，CUDA は分岐予測機能を持たないないため分岐処理に弱く，そのうえ同一 Multiprocessor(MP) 上の各 StreamingProcessor(SP) は同時に同じ種類の演算しか行えないため，計算順序やデータ配置を調整して分岐を減らし (もしくは分岐方向が揃うようにし) 常に同じ種類の演算が行われるようにすることが重要である．さらに，同一 CUDA Block(MP 単位の実行単位) 内の CUDA Thread(SP 単位の実行単位) 同士では高速で低レイテンシだが小容量の SharedMemory が共有できる上に GPU カーネル内で CUDA Thread 間の同期が行える一方で，CUDA Block 同士では SharedMemory が共有できない上に同期機構も用意されていない．(GPU カーネルを終了することで同期する．Atomic 演算を用いて同期機構を作ることも不可能ではないが，性能が期待できないためここでは考慮に入れない．)

櫻井らによるスカラ計算機向けの実装ではマルチコア計算機を用いた実行が想定されており，各コアが並列に各セグメントベクトルを逐次実行することで高性能が得られる．方式の名前にもなっている “分岐の削減” はこの逐次実行を効率的に行うためのアルゴリズム改良である．一方 CUDA については，CPU と比べて大量の計算コアを備えてはいるものの，CPU と同様に各コアが各セグメントベクトルを逐次実行しても高い性能は得られない．これは，入力行列の再利用性がないため高速な SharedMemory を活用しにくいうえに，同一 CUDA Block 内の CUDA Thread 同士が行う処理に多数の分岐処理が必要になったり演算を揃えることが難しくなったりしてしまうためである．

そこで我々の実装においては，まず Segmented Scan 方式においてセグメントベクトル間の計算には並列性があることを利用し，セグメントベクトル単位で CUDA Block に計算を並列実行させている．さらにセグメントベクトル内の計算を CUDA Thread により SharedMemory を用いて並列実行させることにした．実装の詳細は 3.3 節で述べる．

一方で複数セグメントベクトルにまたがった入力行の部分積を統合する処理については，複数セグメントベクトルに連続してまたがった入力行が存在しうるため，単純に並列化することは困難である．そこで，Shubhabrata らの実装を参考にして階層的な Segmented Scan を行うことにした．実装の詳細は 3.4 節で述べる．

3.3 インデックスを用いたリダクション演算

本節では各セグメントベクトル内の計算を CUDA Block 内の CUDA Thread 群に割り当てる方法について述べる．

はじめに単純な割り当てを行う際の問題点について確認する．Segmented Scan 方式におけるセグメントベクトル内のデータ配置および必要な計算について確認してみると，入力行列の各行における非零要素数の配置に依存しており，各要素を対応する入力行列と個別に乗算するだけで良い場合 (非零要素数が 1 の場合) もあれば，乗算の後に入力行列の同一の行に配置されている要素同士での和計算 (リダクション演算) が必要となる場合 (非零要素数が複数の場合) もある．CUDA において SharedMemory を用いた高速な並列リダクション演算を行う方法は知られている．しかし今回のようにリダクション演算を行う必要の有無やリダクション演算の範囲が不定な問題には，単純に並列リダクションを用いて高速化を行うことはできない．

そこで我々はセグメントベクトル内の計算について，インデックスを利用したリダクション演算を提案することによって高速化を行った．図 5 にインデックスを利用したリダクション演算の概要と計算例を示す．なお，本手法はセグメントベクトル長が 2 のべき乗数であることを前提としており，実装においてはパディングなどの処理が必要となることがある．

我々の手法では，セグメントベクトル内の各要素がセグメントベクトルの先頭もしくは入力行列における行頭要素から何番目の要素であるか，という数値 (インデックス) を用意する．一方でカウンタを用意し，初期値としてセグメントベクトル長の半分の値を設定する．そして各要素について，自身のインデックスがカウンタ値以上かつカウンタ値の二倍未満の場合には自身よりカウンタ値分だけインデックスが小さな要素に対して自身の値を加算する，という処理を行う．あとはカウンタの値を半減させるたびに同様の処理を繰り返すことで，セグメントベクトルの先頭要素および入力行列における行頭要素に対するリダクション

© 2010 Information Processing Society of Japan

Vol.2010-HPC-126 No.1
2010/8/3

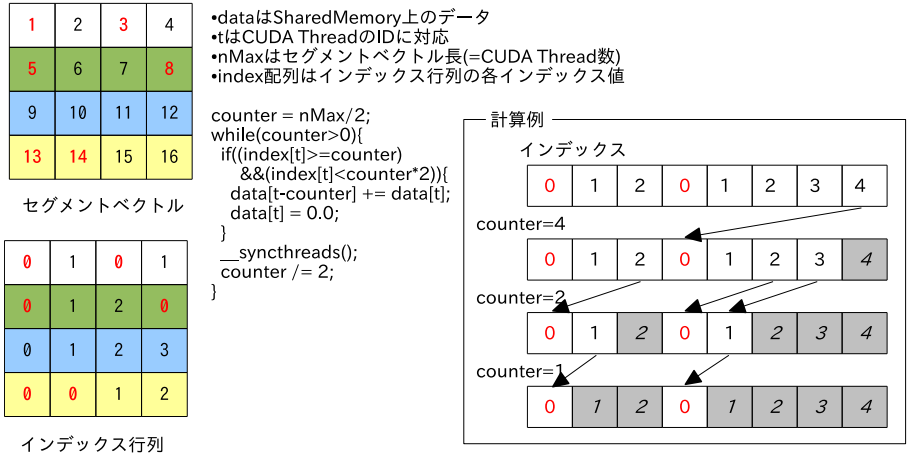

図 5 インデックスを用いたリダクション演算
Fig. 5 Reduction calculation using index

演算が可能となる．

本手法はセグメントベクトルごとに SharedMemory へ値をコピーして計算することが可能であり，またカウンタとインデックス値の比較についても三項演算子を用いた比較および比較結果を利用したダミーライト (計算を阻害しない適当なメモリへの値加算) により CUDA Thread 間の処理を統一させることが可能である．そのため，高速にリダクション演算を行うことが可能である．また SharedMemory 上で計算しているため入力行列の内容も保持しやすい．

本手法のデメリットとなり得る点については，インデックスを用意することに関する時間とメモリ使用量のコストが挙げられる．本手法では各要素に対するインデックス分のメモリが必要である．一方インデックスの作成にかかる時間については，セグメントベクトルごとに全要素を一度走査する必要はあるものの，セグメントベクトル単位で並列に走査できる部分が多いためマルチコア CPU 上で高速に作成することができると考えられる．

3.4 再帰的 Segmented Scan 方式

セグメントベクトルにまたがった部分積の統合処理は，セグメントベクトルの数が少ない場合には逐次処理を行っても全体の実行時間に及ぼす影響は軽微であるが，セグメントベクトルの本数が増えると大きな影響を持ってしまう．今回の我々の実装は，入力行列の持つ非

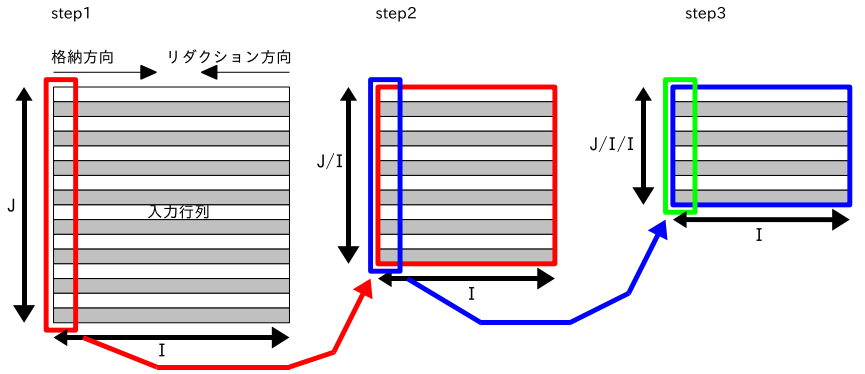

図 6 再帰的な Segmented Scan 方式
Fig. 6 Recursive Segmented Scan method

零要素の数が多い場合にセグメントベクトルの本数が多くなるため，部分積の統合を並列高速化することは重要である．

しかしこの統合処理は，複数のセグメントベクトルにまたがって統合する必要がある場合もあるため，単純に並列化をすることができない．これに対して Shubhabrata らは，統合処理自体も Segmented Scan 処理によって行えることに着目し，再帰的な Segmented Scan 処理を行っている．

そこで，我々の実装においても再帰的な Segmented Scan 処理を行うことにした (図 6)．我々の実装は 3.3 節で述べたように事前にインデックスを計算しており，部分積の統合処理についても入力行列の形状 (非零要素の配置) に依存するため事前にインデックスを作成しておくことが容易である．そのため計算結果を求めるために必要な全てのインデックスを事前に求めておいて利用することにした．

4. 性 能 評 価

4.1 性能評価環境と対象問題

実装の有効性を確認するために性能評価実験を行った．性能評価環境は表 1 の通りである．

対象問題としては，フロリダ大学の Sparse Matrix Collection[8] に含まれる 6 つの非対称行列 (以下，フロリダ行列と呼ぶ) を選出した．また，過度に偏った行列についても実験を行った．各行列の形状 (大きさ，偏りの有無) を表 2 および表 3 に示す．

© 2010 Information Processing Society of Japan

表 1 評価環境
Table 1 Evaluatoin environment

CPU	Intel Xeon W3520 2.67GHz (4 コア，HT 無効)
メインメモリ	12.0GB
GPU	NVIDIA GeForce GTX 285 SP 数 240, SP クロック 1.48GHz
ビデオメモリ	1.0GB
CPU-GPU 接続	PCI-Express Gen2 x16
OS	CentOS 5.4 (kernel 2.6.18)
コンパイラ	GCC 4.1.2, nvcc 3.1 v0.2.1221

表 2 テスト用行列一覧 1: フロリダ行列
Table 2 Test matrices1: Florida Sparse Matrix Collection

	Matrix(行列の名称)	N(一辺の長さ)	NNZ(合計非零要素数)
	poisson3Da	13514	352762
	hcircuit	105676	513072
	Baumann	112211	748331
*	dc2	116835	766396
*	language	399130	1216334
*	rajat29	643994	3760246

(名称の前に*があるものは非零要素の配置の偏りが大きな行列)

比較対象として，シンプルな行単位での並列化，Segmented Scan 方式を用いている GPU(CUDA) 向けライブラリ CUDPP 1.1.1，そして Segmented Scan 方式を用いていない GPU(CUDA) 向けライブラリ CUSP v0.1.0[9] でも性能を測定した．これらの計算はいずれも CPU-GPU 間のデータ転送時間を含んでおらず，単精度浮動小数点演算を行っている．さらに，CPU を用いた場合の参考性能として OpenATLib version Beta における Branchless Segmented Scan 方式での演算性能 (コンパイラとして gfortran 4.1.2 を使用，コンパイルオプション -O3 -fopenmp，OpenMP による 4 スレッド並列実行，倍精度浮動小数点演算) も測定した．

4.2 評価結果

図 7 および図 8 に各入力行列に対する性能を示す．それぞれ縦軸に性能 GFLOPS 値を示しているが，図 8 は問題サイズによる性能差が大きく，さらに各実装の性能差が非常に大きなため対数グラフを用いている．

まずフロリダ行列についての性能 (図 7) について見てみると，Segmented Scan 方式

表 3 テスト用行列一覧 2:過度に偏った行列
Table 3 Test matrices2: biased matrix

Matrix	N	NNZ
mat1K	1000	1999
mat10K	10000	19999
mat100K	100000	199999
mat1M	1000000	1999999

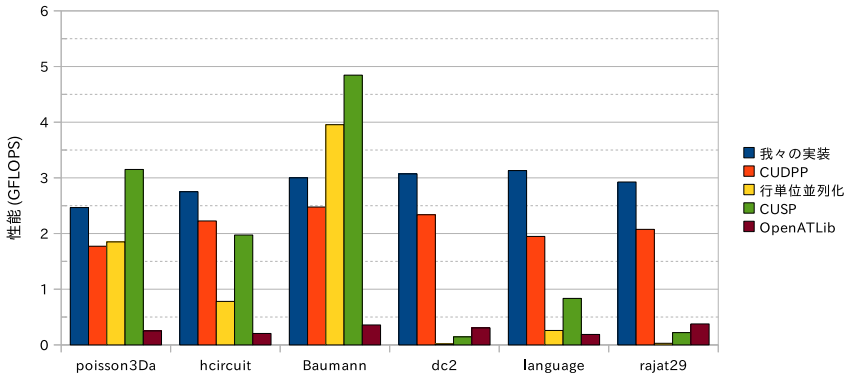

図 7 フロリダ行列での性能
Fig. 7 Performance result of Florida Sparse Matrix Collection

(Branchless Segmented Scan 方式) を用いているものと用いていないものとで性能の傾向に明確な違いが生じていることが確認できる．前者 (我々の実装，CUDPP，OpenATLib) の演算性能は入力行列の形状に対する演算性能の変化が小さい一方，後者 (行単位並列化，CUSP) の演算性能は偏りが大きい行列に対する性能が著しく低い．我々の実装は前者であるため入力行列の形状による演算性能の変化が小さく，また CUDPP に対して常に高い性能を得られていることが確認できる．その一方で，偏りの小さな行列の一部 (Baumann) では我々の実装と CUDPP が行単位並列化よりも低い性能となっており，さらなる性能の向上や，問題形状に応じて Segmented Scan を用いない手法に切り替える機能などの必要性が感じられる．なお，フロリダ行列において我々の実装が最も高い性能を得たのは，language 行列における 3.13GFLOPS である．このときの CUDPP に対する性能比率は 1.61 倍である．

続いて過度に偏った行列の性能 (図 8) について見てみると，フロリダ行列以上に Seg-

© 2010 Information Processing Society of Japan

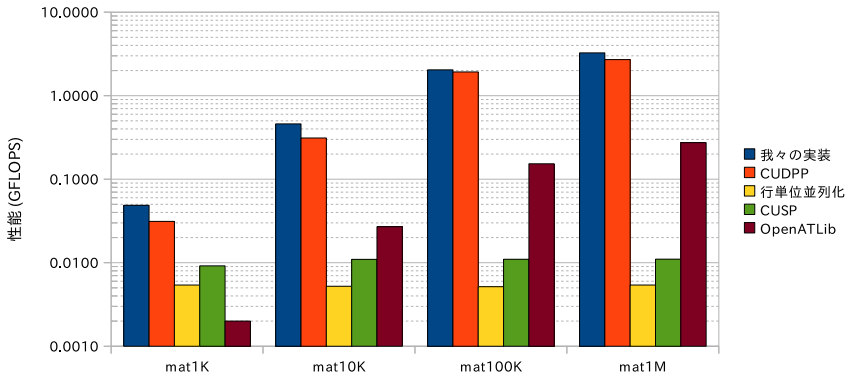

図 8 過度に偏った行列での性能 (対数グラフ)
Fig. 8 Performance result of biased matrix (logarithmic chart)

mented Scan 方式を用いているか否かによる傾向の違いが生じている．Segmented Scan 方式を用いていない行単位並列化と CUSP は非零要素数が大きな場合でも 0.01GFLOPS 程度以下の低い性能しか得られていない一方で，Segmented Scan 方式を用いている実装はより高い性能が得られている．過度に偏った行列において我々の実装が最も高い性能を得たのは，mat1M 行列における 3.26GFLOPS である．このときの CUDPP に対する性能比率は 1.20 倍である．

フロリダ行列と過度に偏った行列全体において我々の実装が最も高い性能を得たのは mat1M 行列における 3.26GFLOPS である．また CUDPP に対する我々の実装の性能比率が最も高かったのは，language 行列における 1.61 倍である．ただし，CUDPP の性能は我々の実装のように事前に行列情報の準備をしてはいない．また，language 行列の実行時間が 0.78msec なのに対して CPU 上での行列情報の準備には 10msec，測定範囲外の CPU-GPU 間データ転送にも 5msec 程度かかっている．行列情報の準備と CPU-GPU 間データ転送についてはさらに最適化の余地があるものの，現状では 1 回の計算に比べて長い時間がかかっている．以上から，実際のアプリケーションにおいては行列情報の準備や CPU-GPU 間の通信時間が表面化しない利用方法をとる必要がある．

5. おわりに

本稿では新たな CUDA 向けの疎行列ベクトル積計算方法を提案し，その実装と性能について述べた．我々の実装は Segmented Scan 法をベースにしており，入力行列の形状情報 (インデックス) を用いたリダクション演算や再帰的な Segmented Scan 処理によって並列高速化を行っている．性能評価の結果，非零要素の配置に偏りのある行列において最大で 3.26GFLOPS の性能を達成した．

今後の展望については，本稿では 256 に固定していたセグメントベクトル長を変更させて SharedMemory 使用量や必要な再帰リダクション段数が変わった際の性能調査と最適パラメタの導出を行う予定である．また，入力行列の形状情報に基づく実行時間の見積もりを検討しており，さらには OpenATLib への組み込み，形状情報をもとに CPU と GPU もしくは我々の実装と他の実装とを切り替えて使用する仕組みの実装も検討している．新アーキテクチャである Fermi を用いた性能評価やさらなる最適化の検討，および倍精度浮動小数点演算を用いた場合の性能についても評価を行う予定である．

謝辞 本研究は文部科学省「e-サイエンス実現のためのシステム統合・連携ソフトウェアの研究開発」，シームレス高生産・構成のプログラミング環境，の支援を受けている．

参 考 文 献

1) gpgpu.org: General-Purpose computation on GPUs(GPGPU), http://gpgpu.org/.
2) 片桐孝洋，櫻井隆雄，黒田久泰， 直野健，中島研吾：OpenATLib:汎用的な自動チューニングインターフェースの設計と実装，情報処理学会研究報告 2009-HPC-121，pp.1–10 (2009).
3) 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).
4) 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).
5) NVIDIA: NVIDIA GPU Computing Developer Home Page, `http://developer.nvidia.com/object/gpucomputing.html`.
6) CUDPP: CUDA Data Parallel Primitives Library, `http://code.google.com/p/cudpp/`.
7) Sengupta, S., Harris, M. and Garland, M.: Efficient Parallel Scan Algorithms for GPUs, Technical Report NVR-2008-003, NVIDIA Corporation (2008).
8) T.A.Davis: Sparse Matrix collection, `http://www.cise.ufl.edu/research/sparse/matrices/`.
9) cusp library: Generic Parallel Algorithms for Sparse Matrix and Graph Computations, http://code.google.com/p/cusp-library/.

© 2010 Information Processing Society of Japan