

GPUにおけるCRS形式疎行列ベクトル積の自動チューニング

吉澤 大樹¹ 高橋 大介²

概要：GPUにおける疎行列ベクトル積の性能は実行時パラメータに大きく依存するため、最適なパラメータを選択して計算することが重要である。本論文では、GPUにおけるCRS形式疎行列ベクトル積の性能が、あるパラメータに大きく依存していることを示し、さらに最適なパラメータを自動選択するためのアルゴリズムを提案する。自動選択アルゴリズムを用いてCRS形式疎行列ベクトル積を自動チューニングすることで、CUSPARSEと比較して最大約1.26倍の性能向上を達成した。また、疎行列ベクトル積の高速化により、連立一次方程式の反復解法である共役勾配法においてCULA Sparseと比較して最大約1.1倍の性能向上を達成した。

1. はじめに

SpMV（疎行列ベクトル積）は、連立一次方程式、固有値問題、偏微分方程式などで用いられる。そのため、SpMVは科学技術計算をはじめとする様々な分野のアプリケーションにおいて極めて重要な計算である。それらのアプリケーションを高速に実行するため、SpMVの高速化には大きな期待が寄せられている。疎行列ベクトル積は計算機環境や計算に用いる疎行列によって最適な計算手法が異なる。そのため疎行列ベクトル積の高速化手法が今までに数多く提案されている [1][2][3][4]。

一方、グラフィック演算に用いられるGPU（Graphics Processing Unit）を汎用的な計算に用いるGPGPU（General Purpose computation on GPU）が普及してきている。GPUは多数のプロセッサコアが搭載されたメニーコアプロセッサであり、高い並列演算性能とメモリバンド幅、コストパフォーマンスの高さが特徴である。SpMVは演算量に対するメモリアクセス量が多い計算であるため、メモリバンド幅に律速される。また、入力ベクトルに対するメモリアクセスパターンが不規則になるため、メモリアクセス速度の低下を招きやすい。したがってメモリアクセスパターンを改善することによりGPUの大きいバンド幅を活かし、SpMVの性能を向上させることができる。

疎行列を格納する方法の違いにより、SpMV計算時のメモリアクセスパターンが大幅に異なるため、最適なメモリアクセスパターンを実現するために、数々の格納形式が提案されている [5][6][7]。しかしながら、SpMVを高速に計算するために最も適した格納形式は、計算機の種類や計算に用いる疎行列の性質によって異なる。また、各格納形式におけるSpMVの性能差が大きいため、最適な格納形式を選択することによるSpMVの性能向上効果は大きい [8]。

各々のスレッドに対してどのようにタスクを分割して割り当てるかを定める、スレッドマッピング手法もSpMVの性能を向上させる上で非常に重要である。

SpMVにおいては、主に行単位や非零要素単位でタスクを分割し、スレッドに割り当てることでスレッドマッピングが行われる。スレッドマッピングはメモリアクセスパターンに直接影響するため、あらゆる格納形式のSpMVにおいて十分な性能を得るためには最適なスレッドマッピングが必要となる。

本論文では、CRS（Compressed Row Storage）形式と呼ばれる格納形式のSpMVにおいて、スレッドマッピングを自動最適化することで高速化する。スレッドマッピングを最適化するため、実行時パラメータを自動選択するアルゴリズムを提案する。実行時パラメータの自動選択アルゴリズムを適用することでCRS形式SpMVの性能が向上することを示す。さらに、実アプリケーションへの応用として、疎行列連立一次方程式の反復解法を実装し、性能を評価することで、本研究で実装した自動チューニングの効果を検証する。

¹ 筑波大学大学院システム情報工学研究科
Graduate School of Systems and Information Engineering,
University of Tsukuba

² 筑波大学システム情報系
Faculty of Engineering, Information and Systems, University
of Tsukuba

2. 関連研究

Vázquez らの研究 [1] では、GPU 上の ELLR-T と呼ばれる SpMV のアルゴリズムにおいて、高速化を行っている。ELLR-T は ELLPACK と呼ばれる格納形式を用いた SpMV のアルゴリズムである。ELLR-T では複数の実行時のパラメータによって性能が大きく変化することが知られている。Vázquez らの研究では、これらのパラメータを自動的に選択することで SpMV の高速化を行っている。自動選択のためのパラメータには 1 行ごとの非零要素数の平均などが用いられるため、実行前に疎行列全体を調べる必要がある。本論文でも、事前に疎行列の特徴を調べた上で自動チューニングを行う。

Kubota らの研究 [2] では、疎行列の最適な格納形式を自動的に選択し、その格納形式へ変換することで GPU 上での SpMV の高速化を行っている。格納形式を自動選択するために、予備評価として事前に複数の格納形式に関して SpMV の性能を評価する。その予備評価をもとに自動選択に必要なパラメータを決定し、格納形式の自動選択を行う。

これらの研究では、予備評価で定められたパラメータを用いて自動チューニングを行うオフライン自動チューニングの手法を用いている。

Cevahir らの研究 [3] では、GPU クラスタ上での共役勾配法を解く際に SpMV の最適化を行っている。共役勾配法を解く前に、複数の格納形式で SpMV を数回実行し、最適な格納形式を選択する。この研究ではオンライン自動チューニングの手法を用いている。オンライン自動チューニングでは予備評価などを行わず、実行時に複数候補に対して一定の試行を行うことでチューニングを行う。ここで試行の処理を全体の処理と比較して小さくできれば、チューニングにかかる時間は大きなオーバーヘッドにならない。

3. CRS 形式 SpMV

A を疎行列、 x を入力ベクトル、 y を出力ベクトルとしたときの、 $y = Ax$ ($A \in \mathbb{R}^{n \times n}$, $x \in \mathbb{R}^n$, $y \in \mathbb{R}^n$) を疎行列ベクトル積とする。

3.1 疎行列格納形式

疎行列は行列中の零要素の割合が多いため、零要素を省くなど圧縮して格納される。格納方法のアルゴリズムの違いによって様々な格納形式が提案されている。本章では、疎行列格納形式の一つである CRS 形式について述べる。

CRS 形式は疎行列を行方向に走査し、非零要素を格納する格納形式である。以下、 $n \times n$ 行列 $A = (a_{ij})$ の非零要素の総数を nnz とする。CRS 形式は図 1 に示すように、3 つの配列から構成される。(1) 配列 val 、非零要素の値を格納

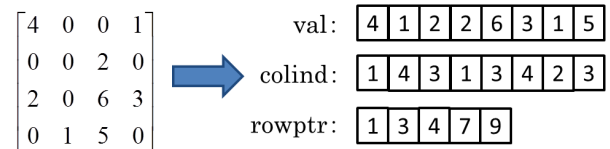


図 1 CRS 形式を構成する配列

する、長さ nnz の配列。(2) 配列 $colind$ 、配列 val に格納された非零要素に対する列番号を格納する、長さ nnz の配列。(3) 配列 $rowptr$ 、配列 val と $colind$ における各行の先頭を表す、長さ $n+1$ の配列。配列の最後に非零要素数 - 1 の値を格納する。

3.2 スレッドマッピング

CRS 形式 SpMV のスレッドマッピング方式で最も基本的な方式として行分割方式がある。行分割方式では、疎行列を行単位で分割し、各行に対して 1 スレッドが計算を担当する。

行分割方式を拡張したスレッドマッピング方式として非零要素分割方式がある。非零要素分割方式では、行分割方式同様に疎行列を行単位に分割するが、各スレッドの担当する計算量がなるべく均等になるように、複数行の計算を 1 スレッドが担当する場合がある。

上記の 2 つのアルゴリズムでは GPU のコアを十分に使い切ることが難しい。そのため、GPU 上の実装では 1 行の計算を複数スレッドが担当することで GPU のコアを使い切り、GPU の性能を引き出すことができる。このように 1 行ごとに複数スレッドが計算を担当する CRS 形式 SpMV のアルゴリズムとして、CRS-vector [9] がある。CRS-vector では、1 行ごとに 32 スレッドが計算を担当する。また、SpMV4GPU [10] の CRS 形式 SpMV のスレッドマッピング方式では、1 行ごとに 16 スレッドが計算を担当するようにマッピングを行っている。これらのスレッドマッピング方式のように、1 行あたりの計算を T 個のスレッドが担当する CRS 形式 SpMV のスレッドマッピング方式を以下 CRS-T 方式とする。ここで、 $T = 1$ の CRS-T は行分割方式のスレッドマッピング方式に相当する。

CRS-T 方式 SpMV では、 T 個のスレッドが i 行目における出力ベクトルの要素 $y[i]$ を計算する場合、各スレッドが計算した値 $y[i]_p$ は、部分的な計算結果として一時的にシェアードメモリに格納される。全てのスレッドの部分計算が完了した後、 $y[i] = \sum_{p=1}^T y[i]_p$ によって i 行目における出力ベクトルの要素を求める。 $y[i] = \sum_{p=1}^T y[i]_p$ の計算は、 T 個のスレッドを用いて木構造的に行われる。このように、木構造的に複数の要素から一つの値を求める処理をリダクションと呼ぶ。CRS-T 方式 SpMV においては、スレッド数 T を適切に選択することで性能を向上させることができる。しかし、最適な T の値は計算に用いる疎行列に

```

int thread_id = blockDim.x * blockIdx.x + threadIdx.x;
int lane = thread_id & (T - 1);
int row = thread_id / T;

if(row < num_rows){
    int row_start = rowptr[row]-1;
    int row_end = rowptr[row+1]-1;

    // compute running sum per thread
    int jj;
    vals[threadIdx.x] = 0;
    for(jj = row_start+lane; jj < row_end; jj += T){
        vals[threadIdx.x] += data[jj] * x[colind[jj]-1];

    // parallel reduction in shared memory
    if(lane < 32) vals[threadIdx.x] += vals[threadIdx.x+16];
    if(lane < 16) vals[threadIdx.x] += vals[threadIdx.x+8];
    if(lane < 8)  vals[threadIdx.x] += vals[threadIdx.x+4];
    if(lane < 4)  vals[threadIdx.x] += vals[threadIdx.x+2];
    if(lane < 2)  vals[threadIdx.x] += vals[threadIdx.x+1];

    // first thread writes the result
    if(lane == 0) y[row] += vals[threadIdx.x];
}

```

図 2 $T = 32$ における CRS-T 方式 SpMV の GPU カーネルコード

表 1 評価環境

CPU	AMD Opteron 6134 2.3GHz 8 cores×2
メインメモリ	DDR3 SDRAM 1333MHz 16GB
GPU	NVIDIA Tesla C2050 1.15GHz 448 CUDA cores
ビデオメモリ	GDDR5 SDRAM 384bit 1.5GHz 3GB (ECC on)
理論ピーク性能	515 GFlops (倍精度)
Compiler	nvcc 4.1 (-O3 -arch=sm_20)

よって異なるため，事前に最適な T の値を予測する必要がある．

そこで本研究では CRS-T 方式 SpMV の T を自動的に選択し計算を行う自動チューニングの実装を行った． $T = 32$ における CRS-T 方式 SpMV の GPU カーネルコードを図 2 に示す．4 章では， $T = 1, 2, 4, 8, 16, 32$ における CRS-T 方式 SpMV の予備評価を行う．さらに評価結果を解析し，CRS-T 方式 SpMV の性能が最も良くなるようなスレッド数 T を自動的に求めるためのモデルを定義する．

4. CRS-T 方式 SpMV の最適化

4.1 予備評価

本章では，CRS-T 方式 SpMV で 1 行あたりの計算を行うスレッド数 T を自動選択するためのパラメータを決定するために予備評価を行う．評価環境は表 1 の通りである．予備評価に用いる疎行列は，The University of Florida

表 2 評価に用いた行列の一部

疎行列の名前	1 辺の長さ	非零要素数	$rlmax$
ecology	1000000	4996000	5
ahache2	715176	4817870	8
tmt_sym	726713	5080961	9
qa8fm	66127	1660579	27
af_shell9	504855	17588845	40
s3dkt3m2	90449	3686223	42
pwtck	217918	11634424	90
F1	343791	26837113	378
nd24k	72000	28715634	483

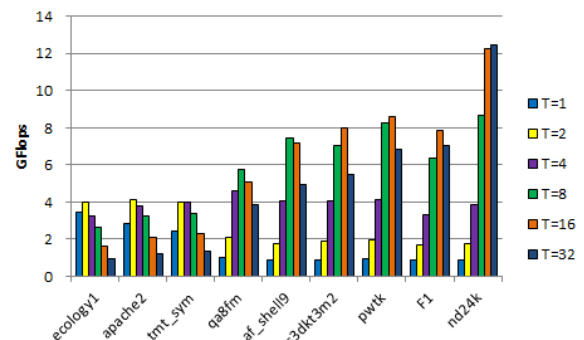


図 3 各スレッド数における CRS-T 方式 SpMV の性能

Sparse Matrix Collection [11] から入手した 17 個の疎行列を用いる．任意の疎行列に対して， $T = 1, 2, 4, 8, 16, 32$ における CRS-T 方式 SpMV の性能評価を行う．性能評価には倍精度の Flops 値を用いる．SpMV の Flops 値は，疎行列の非零要素数を nnz ，計算に要した時間を $time$ [sec] として， $nnz \times 2 / time$ となる．予備評価を行った疎行列のうち，一部の疎行列に対する評価結果を図 3 に示す．また，それらの疎行列に関する情報を表 2 に示す．

4.2 自動選択アルゴリズム

予備評価の結果をもとに，最適なスレッド数を自動選択するための判別式を定義する．疎行列の 1 行あたりの非零要素数の最大値 ($rlmax$) が大きくなるほど最適なスレッド数の値も大きくなる傾向がある．また，各スレッドが行う行列要素とベクトル要素の積和演算が 2~4 回の間になるように 1 行あたりのスレッド数 T を決定すると，多くの場合で CRS-T 方式 SpMV の性能が最適化される． $rlmax$ が 32 以上の疎行列に関しては， $T = 16$ で CRS-T 方式 SpMV の性能が最適になることが多く， $T = 32$ で最適となる場合でも $T = 16$ と $T = 32$ の CRS-T 方式 SpMV の性能比は 1.02 以下にとどまる．

以上の結果より，スレッド数自動選択のための判別式を $rlmax$ を用いて下記の式で定義する．以下，判別式により求まる T の値を T^* とする．

$$T^* = \begin{cases} 16 & (rlmax \geq 32) \\ 2^{\lceil \log_2 rlmax \rceil - 2} & (rlmax < 32) \end{cases} \quad (1)$$

上記の結果は，CRS-T 方式 SpMV において $y[i]_p$ を求めるコストとリダクションにより出力ベクトルを求めるコストの関係からも予測できる．CRS-T 方式 SpMV において， T を大きくするほど，1 スレッドが計算する要素数は減少するため， $y[i]_p$ を求める処理時間は短縮される．一方，リダクションは木構造的に要素の和を求めるため， 2^n 個の要素から和を求める場合は n ステップの処理が必要になる．CRS-T 方式 SpMV の場合，リダクションでは T 個の要素から和を求めるため， T を大きくするほど，リダクションに費やされる時間が延びる．

1 行あたりの非零要素数が十分に大きい疎行列の場合， $T \leq 16$ ならば， T を大きくするほど CRS-T 方式 SpMV のパフォーマンスは向上する． $y[i]_p$ を求める処理部分で，レイテンシの大きいグローバルメモリへのメモリアクセスを多く行っているのに対し，リダクション処理部分では，大半がレイテンシの小さいシェアードメモリへのメモリアクセスである．そのため， T の増大によって各スレッドにおけるグローバルメモリへのアクセス回数が減少し，シェアードメモリへのアクセス回数が増加する．結果として，SpMV 全体でのメモリアクセスのレイテンシを減少させることができる．

$T = 32$ の CRS-T 方式 SpMV では， $T \leq 16$ の場合には考慮する必要がなかった問題が発生する．GPU では 32 個のスレッドがウォープという単位で管理され，SIMD 命令により動作するが，メモリアクセスに関しては必ずハーフウォープ単位で実行される．そのため，32 個のスレッドに対して同期が必要になるようなプログラムでは，メモリアクセスを行う際，ハーフウォープ単位で 2 回のメモリアクセスが行われ，この 2 つのメモリアクセスに対して暗黙の同期が取られる．SpMV はメモリ律速な計算であるため，メモリアクセスの同期によるコストの増加がパフォーマンスの低下を引き起こしやすい．

本研究で実装を行わなかった $T \geq 64$ の CRS-T 方式 SpMV では， $T = 32$ の場合に起こる暗黙の同期以外に，明示的な同期をとる必要がある．なぜなら，32 個より多いスレッドに対して同期が必要なプログラムでは，複数ウォープ間で同期をとる必要があるためである．その場合 CUDA のプログラミングモデルでは，ソースコード中に同期をとる命令を記述しなければならない．この命令で行われる同期は，ブロック中全てのスレッドに対して行われる．そのため，明示的な同期は暗黙の同期に比べてコストが大きく， $T \geq 64$ の CRS-T 方式 SpMV では $T \leq 32$ の場合と比較してパフォーマンスが低下すると予想される．

1 行あたりの非零要素数が少ない場合，例えば 1 行に 4 つの非零要素を持つ疎行列に対して CRS-T 方式 SpMV を

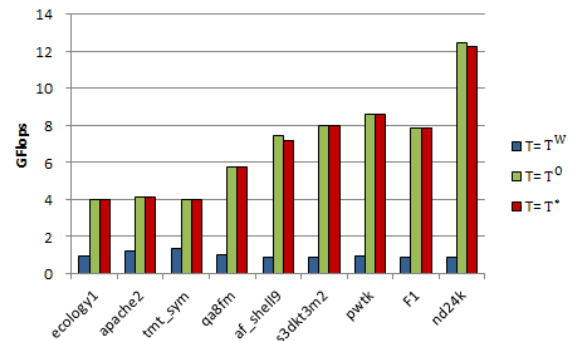


図 4 $T = T^*, T^O, T^W$ における CRS-T 方式 SpMV の性能評価

行うとき， T を 4 より大きくしても $y[i]_p$ を求める処理のコストは減少しない．一方で T の増大により，リダクション処理のコストは増加するため，1 行あたりの非零要素数よりも T を大きくすると，CRS-T 方式 SpMV のパフォーマンスは低下する．

5. 性能評価

評価に用いる疎行列，評価環境，性能の評価基準は，予備評価のときと同様のものを用いる．CRS-T* は， $T = T^*$ における CRS-T 方式 SpMV であり， T^* は 3 章で定義した判別式により決定する．実行時間の計測には SpMV カーネルの実行時間のみを用い，GPU 上でのメモリ領域の確保や CPU と GPU 間のデータ転送時間は考慮しない．これは反復法など，同一疎行列に対して SpMV を複数回行うアプリケーションで用いられることを前提としているためである．また，CRS-T* に関しては疎行列入力と同時に 1 行あたりの非零要素数の最大値 $rlmax$ を求める．そのため， $rlmax$ を求めるオーバーヘッドが小さく，自動チューニングに関するオーバーヘッドは無視できる．

5.1 自動選択アルゴリズムの評価

はじめに自動選択アルゴリズムの評価を行う．CRS-T 方式 SpMV において，最も良い性能と最も悪い性能を出すような 1 行あたりの計算を行うスレッド数 T をそれぞれ T^O, T^W とする．また， $T = T^O, T^W$ における CRS-T 方式 SpMV をそれぞれ CRS-T^O，CRS-T^W とする．図 4 に，CRS-T*，CRS-T^O，CRS-T^W の評価結果を示す．

今回の評価として用いた 17 個の疎行列のうち，14 個の疎行列に対して CRS-T* の性能が CRS-T^O の性能と一致することを確認した．評価に用いた疎行列全てにおいて，CRS-T* は CRS-T^O の約 0.97～1.00 倍の性能が得られた．CRS-T^O は CRS-T^W と比較して約 2.7～15 倍の性能を得られたことから，1 行あたりのスレッド数を変化させることで，CRS-T 方式 SpMV の性能が大幅に変化することを示した．

自動選択アルゴリズムを用いることで，高い精度で最適

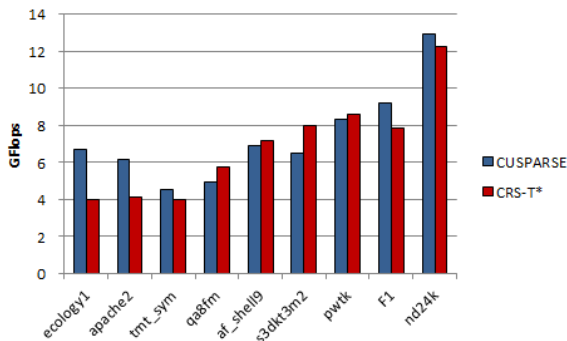


図 5 CRS-T*, CUSPARSE の性能評価

なスレッド数を選択できた。また、CRS-T 方式 SpMV を、本論文で提案した方法により自動チューニングすることで、最適なスレッド数を選択した場合の CRS-T 方式 SpMV と同等程度の性能を得ることができた。CRS-T 方式 SpMV において、スレッド数 T が性能に大きく依存することを示した。

5.2 CRS-T 方式 SpMV の性能に関する評価

CRS-T* と CUSPARSE [5] の CRS 形式 SpMV (以下、CUSPARSE とする) との性能比較を行う。CUSPARSE は NVIDIA が提供している GPU 向けの疎行列計算ライブラリである。性能評価を行った疎行列のうち、一部の疎行列に対する評価結果を図 5 に示す。

評価結果より、一部の行列で CRS-T* が CUSPARSE の性能を上回ったことを確認した。また、CRS-T* は CUSPARSE と比較して、最大で約 1.26 倍の高速化を達成した。

6. 反復解法への応用

SpMV を用いたアプリケーションの 1 つに連立一次方程式の反復解法が挙げられる。反復解法では、SpMV が全体処理の大半を占め、SpMV の高速化がアプリケーション全体の高速化に結びつきやすい。本研究では、GPU 上での前処理なし CG 法 (共役勾配法) を実装し、SpMV 部分を CRS-T 方式 SpMV で自動チューニングすることで高速化する。

CRS-T* を用いた共役勾配法の実装方法を図 6 に示す。図 6 において、スカラー変数はギリシャ文字で表し、ベクトルは太字のローマ字で表す、 A は疎行列を表す。太文字で記述した関数は GPU 上で計算を行うカーネルである。CRS-T* 関数は、自動チューニングを施した CRS-T 方式 SpMV である。DotProduct 関数はリダクションを用いて 2 つのベクトルの内積を計算する。VectorSum 関数は 2 つのベクトルの和を計算する。解ベクトルの各要素はそれぞれ別のスレッドで計算される。

```

r0 = b
p0 = b
γ1 = DotProduct(r0, r0)
for k = 0 to iterLimit do
    qk = CRS-T*(A, pk)
    γ2 = DotProduct(pk, qk)
    α = γ1/γ2
    xk+1 = VectorSum(xk, αpk)
    rk+1 = VectorSum(rk, -αqk)
    γ2 = DotProduct(rk+1, rk+1)
    β = γ2/γ1
    γ1 = γ2
    pk+1 = VectorSum(rk+1, βpk)
    σ = sqrt(γ2)
    if σ < ε then
        break
done

```

図 6 CRS-T* を用いた共役勾配法の実装方法

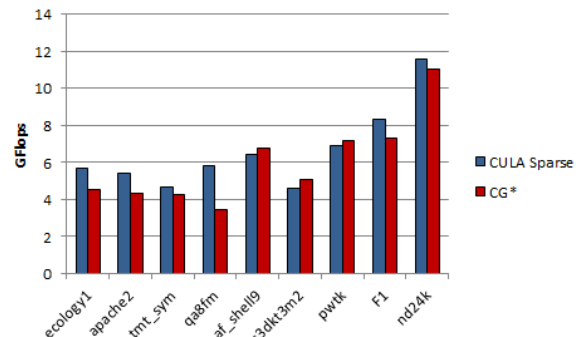


図 7 CRS-T* を用いた CG 法と CULA Sparse CG 法の性能

6.1 性能評価

SpMV に CRS-T* を用いた CG 法 (以下、CG* と表す) と、CULA Sparse [12] の CG 法 (以下、CULA Sparse と表す) の性能を評価し比較を行う。CULA Sparse は EM Photonics が提供している反復解法ライブラリであり、SpMV 部分は CUSPARSE の API によって実装されている。

疎行列の格納形式は CRS 形式を用い、前処理は行わない。CG 法は 1 回の反復で、SpMV 1 回と 5 回の Level-1 BLAS (Basic Linear Algebra Subprograms) 関数が実行されるため、CG 法の Flops 値は、疎行列の非零要素数 nnz 、行列サイズ n 、反復回数 $iteration$ 、収束までに要した時間 $time$ を用いて、 $iteration \times (2 \times nnz + 10 \times n) / time$ となる。測定結果を図 7 に示す。

CG* は CULA Sparse と比較して、最大約 1.1 倍の性能向上が得られた。

CRS-T* が CUSPARSE の性能を上回っているにもかかわらず、CG* が CULA Sparse の性能を下回るような疎行列が存在したことから、SpMV 以外の処理では CULA Sparse が CG* の性能を上回っていると考えられる。今後 CG* における SpMV 以外の処理を改良することで、さらなる性能向上が期待できる。

7. まとめ

本研究では、オフライン自動チューニングにより CRS-T 方式 SpMV の高速化を行った。1 行あたり T 個のスレッドが計算を担当する CRS-T 方式 SpMV の性能を解析することで、CRS-T 方式 SpMV の性能を最適化するスレッド数の値が、疎行列の 1 行あたりの非零要素数の最大値 $rlmax$ に依存することを示した。また、 $rlmax$ を用いてスレッド数を自動選択するための判別式を定義した。

判別式によって決定したスレッド数 T^* をもとに、CRS-T 方式 SpMV におけるスレッド数の自動選択アルゴリズムを実装し、自動チューニングによる CRS 形式 SpMV の高速化を行った。CRS-T 方式 SpMV を自動チューニングした CRS-T* は、最適なスレッド数を用いた CRS-T 方式 SpMV の約 0.97 ~ 1.00 倍の性能が得られた。

CRS-T* は、疎行列計算ライブラリ CUSPARSE の CRS 形式 SpMV との比較において、一部の疎行列に関して性能が上回り、最大で約 1.26 倍の高速化を達成した。

実アプリケーションへの応用例として、CRS-T* を用いた CG 法の実装を行った。CRS-T* を用いた CG 法は、CULA Sparse と比較して最大約 1.1 倍の高速化を達成し、さらに今後の性能向上の可能性を示した。

今後の課題として、自動選択アルゴリズムの改良による、スレッド数自動選択の高精度化が挙げられる。今回の自動選択アルゴリズムでは、最適なスレッド数を選択できなかった疎行列があったため、1 行あたりの非零要素数の最大値 $rlmax$ 以外の疎行列の特徴を解析することで、パラメータ決定のための変数を増やし、パラメータ決定判別式の精度を高めたい。

謝辞 本研究の一部は、JST CREST「進化的アプローチによる超並列複合システム向け開発環境の創出」による。

参考文献

- [1] Vázquez, F., Fernández, J. J. and Garzón, E. M.: Automatic tuning of the sparse matrix vector product on GPUs based on the ELLR-T approach, *Parallel Computing*, Vol. 38, pp. 408–420 (2012).
- [2] Kubota, Y. and Takahashi, D.: Optimization of Sparse Matrix-Vector Multiplication by Auto Selecting Storage Schemes on GPU, *Proc. 11th International Conference on Computational Science and Its Applications (ICCSA 2011), Part II*, Lecture Notes in Computer Science, No. 6783, Springer-Verlag, pp. 547–561 (2011).
- [3] Cevahir, A., Nukada, A. and Matsuoka, S.: CG on GPU-enhanced Clusters, 情報処理学会研究報告, Vol. 2009-HPC-123, No. 15 (2009).
- [4] 梶山民人, 額田彰, 須田礼仁, 長谷川秀彦, 西田晃: SILC: 行列計算ライブラリの利用を簡単化するフレームワーク, 第 10 回日本計算工学会講演会論文集, pp. 239–242 (2005).
- [5] NVIDIA: CUSPARSE Library User Guide. http://developer.download.nvidia.com/compute/DevZone/docs/html/CUDALibraries/doc/CUSPARSE_

Library.pdf.

- [6] Karakasis, V., Goumas, G. and Koziris, N.: Performance Models for Blocked Sparse Matrix-Vector Multiplication Kernels, *Proc. 2009 International Conference on Parallel Processing (ICPP '09)*, pp. 356–364 (2009).
- [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, Vol. 2010-HPC-126, No. 1 (2010).
- [8] 片桐孝洋, 佐藤雅彦: 疎行列-ベクトル積における実行時データ変換のための自動チューニング方式, 情報処理学会研究報告, Vol. 2011-HPC-130, No. 41 (2011).
- [9] Bell, N. and Garland, M.: Efficient Sparse Matrix-Vector Multiplication on CUDA, NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation (2008).
- [10] Bordawekar, R. and Baskaran, M. M.: Optimizing Sparse Matrix-Vector Multiplication on GPUs, *Engineering*, Vol. 24704, No. RC24704 (2008).
- [11] Davis, T. and Hu, Y.: The University of Florida Sparse Matrix Collection. <http://www.cise.ufl.edu/research/sparse/matrices/>.
- [12] Photonics, E.: CULA Sparse. <http://www.culatools.com/sparse/>.