Vol.2015-HPC-148 No.3
2015/3/2

動的な並列実行機構を用いたSpMV実装の性能評価

大島 聡史[1,a)] 片桐 孝洋[1] 櫻井 隆雄[2] 中島 研吾[1] 黒田 久泰[5] 直野 健[3] 猪貝 光祥[4]

概要：本稿では疎行列ベクトル積 (SpMV) の実装について述べる．疎行列ベクトル積の高速化に関する研究は多く実施されており，行列の形状 (非ゼロ要素の配置) や行列格納形式，実行するハードウェアなど様々な観点からの研究が進められている．我々が本稿において着目するのは，非ゼロ要素の配置に偏りがある疎行列に対して，動的な並列実行機構を活用して高速化を行うことである．現在の OpenMP や CUDA には負荷バランスの悪い並列計算問題に対して性能改善を行える可能性のある動的な並列実行機構が備わっている．そこで，これらの機構を活用することで疎行列ベクトル積の性能を向上させることを目的として実装と性能評価を行った．実験の結果，非ゼロ要素の配置に偏りがあるシンプルな疎行列に対しては高い性能向上が得られた．その一方で，より一般的な行列を用いた場合の効果は限定的であり，むしろ実行時間が大きく伸びる例も多く，実用とするためにはさらなる最適化が必要であることも確認できた．性能向上を阻害する主な原因は動的な並列実行機構のオーバーヘッドにもあると考えられ，今後のハードウェアやシステムソフトウェアの改善も期待される．

1. はじめに

疎行列ソルバーや疎行列ベクトル積 (sparse matrix vector multiplication, 以下 SpMV) は，様々な科学技術アプリケーションにおいて実行時間の多くを占める問題であるため高速化が求められている．これらの計算に関する研究はすでに多く実施されているものの，メモリへの間接アクセスやランダムアクセスが必要となることやキャッシュサイズを越えるデータセットを要求されることが多く高性能を得ることが難しいこと，また並列計算ハードウェアの多様化が進んでいることから，現在も多くの研究が実施されている．

著者らも疎行列ソルバーや SpMV の高速化に多大な興味を持っており，これまでにも幾つかの研究報告や論文を公開している．本稿では特に，近年の並列化プログラミング環境の備える**動的な並列実行機構**を用いた SpMV 実装に着目する．具体的には，マルチコア CPU やメニーコアプロセッサ向けの向け並列化プログラミング環境である OpenMP において利用可能なタスク並列化機構，および GPU 向け並列化プログラミング環境である CUDA において利用可能な dynamic paralleisim 機構を用いて高速な SpMV の実装を目指す．なお，本稿では特に断りの無い限り，CUDA に対応した GPU を単に GPU と呼ぶ．

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 SpMV と既存の SpMV に関する研究について述べる．3 章では実行環境と対象行列について述べる．4 章では OpenMP のタスク並列化機構を用いた SpMV 実装について，また 5 章では CUDA の dynamic parallelism 機構を用いた SpMV 実装について，実装の内容と性能評価結果を述べる．6 章はまとめの章とする．

2. 疎行列ベクトル積 (SpMV)

2.1 疎行列ベクトル積と疎行列格納形式

疎行列ベクトル積 (SpMV) は，疎行列，すなわちゼロ要素の多い行列とベクトルとの乗算問題である．SpMV は CG(Conjugate Gradient) 法をはじめとした連立一次方程式の数値解法など様々な計算に用いられており，アプリケーション実行時間の多くを占めることが多い計算の 1 つとして知られている．そのため高速化の要求が高く，SpMV に関する研究は数多く実施されている [1], [2], [3], [4], [5]．著者らもこれまでに様々なハードウェアや疎行列を対象として SpMV に関する研究を行ってきた [6], [7].

多くの疎行列計算においては大規模な疎行列 (行数や列数の大きな疎行列) を扱いたいという需要が大きい．しかしゼロ要素の多い疎行列を密行列向けのデータ構造を用いて扱おうとすると，ゼロを保存するだけのために大容量のメモリが必要となる．そのうえ SpMV のような計算にお

[1] 東京大学 情報基盤センター
[2] 日立 中央研究所
[3] 日立 横浜研究所
[4] 日立 超 LSI
[5] 愛媛大学 理工学系研究科
[a)] ohshima@cc.u-tokyo.ac.jp

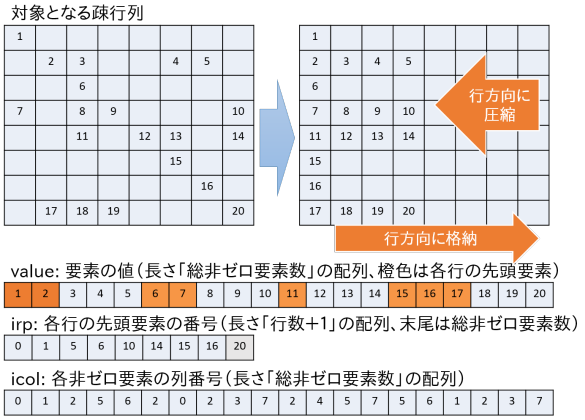

図 1 CRS 形式を用いた疎行列の例

いてはゼロ要素を計算する必要がないため，疎行列のために確保されたメモリ量に対する計算に本質的に必要なメモリ量の割合は小さくなる．そのため疎行列の格納にはメモリ容量効率や計算効率の良い疎行列向けのデータ格納形式が用いられている．特に多く用いられている疎行列格納形式としては，CRS(Compressed Row Storage) 形式 (CSR 形式とも呼ばれる)，CCS(Compressed Column Storage) 形式 (CSC 形式とも呼ばれる)，COO(COOrdinate) 形式，ELL(ELLpack) 形式などがあげられる．さらにこれらの疎行列格納形式をブロック化した形式などのバリエーションも存在している．これらの疎行列格納形式にはそれぞれ適したハードウェアアーキテクチャや計算の種類などがあるため，一概にどの方式が優れたものであるといった判断は困難であり，計算環境や対象問題などにより使い分けられている．

本稿では CRS 形式を対象の疎行列格納形式とする．CRS 形式による疎行列の格納イメージを**図 1** に示す．CRS 形式は非ゼロ要素を行方向に詰めた疎行列格納形式である．CRS 形式の疎行列データは詰められた非ゼロ要素 (value) に加えて，各行の先頭要素をの番号 (irp) と各非ゼロ要素の列番号情報 (icol) から構成される．CRS 形式は余計なゼロ要素を保持する必要がないためメモリ効率が良く，格納可能な疎行列の非ゼロ要素配置に制限もない (特定の行列形状において大きく不利となることがない) ため，多くのアプリケーションやライブラリにおいて標準的な格納形式の 1 つとして利用されている．

2.2 OpenMP を用いた SpMV の並列化

CRS 形式を用いた疎行列に対する SpMV の並列化は容易である．具体的な実現方法としては，SpMV の計算は行ごとに独立して実行可能であるのに対して CRS 形式が行単位で非ゼロ要素を格納する形式であるため，素直に各行に対する計算を並列に実行すれば良い．これは OpenMP のループ並列化指示文によって容易に実現可能である．

CRS 形式を用いた SpMV の典型的な OpenMP プログラム (ソースコード) 例を**図 2** に示す．プログラム全体は二重のループによって構成されており，外側のループは行ごとの計算に，内側のループは行内の計算に対応する．上述の通り CRS 形式を用いた SpMV は行ごとに独立して実行可能であるため，図 2 に示すように外側ループをループ並列化指示文により並列化することが可能であり，多くの場合には妥当な性能を得ることができる．ただし，SpMV は間接メモリ参照を多数繰り返す問題であり，計算性能よりもメモリアクセス性能により律速される種類の問題であることから，OpenMP を用いても n コア利用時に 1 コア利用時の n 倍に近い性能を得られるとは限らない．

```
int i;
#pragma omp parallel for
for(i=0; i<nline; i++){
  int j;
  double tmp = (double)0.0;
  for(j=irp[i]; j<irp[i+1]; j++){
    tmp += value[j] * vector[icol[j]];
  }
  answer[i] = tmp;
}
```

図 2 CRS 形式の疎行列に対する SpMV の OpenMP を用いた並列化実装例

ところで，疎行列における行数や総非ゼロ要素数，行あたりの非ゼロ要素数は対象とする問題 (疎行列) によって大きく異なる．特に総非ゼロ要素数の大小は総演算数や総メモリアクセス数に直結するため実行時間 (性能) を大きく左右するパラメタであり，また行数は並列度と結びつきやすいため並列化効率に大きな影響を与えうるパラメタである．一方で行あたり非ゼロ要素数の偏りも実行時間，特に OpenMP 並列化時の実行時間に大きな影響を与えることがある．ここで，行あたりの非ゼロ要素数が均等に近い疎行列を**バランスの良い行列**，そうでない疎行列を**バランスの悪い行列**と呼ぶことにする．バランスの悪い行列の極端かつわかりやすい例としては，行数 100 の疎行列において，第 1 行のみが 100 の非ゼロ要素を持ち，他の 99 行が 1 の非ゼロ要素を持っているような行列が考えられる．このようなバランスの悪い行列を対象として SpMV の並列化を行う場合，行単位の OpenMP 並列化では第 1 行を担当するスレッドだけが他のスレッドよりも多くの実行時間を必要とし，他のスレッド群は残りの 99 行を計算し終えても第 1 行の計算終了を待たねばらなず，全体の実行時間はあまり短くならない．そのため，たとえ同じ総非ゼロ要素数を持つ疎行列であっても，非ゼロ要素のバランスにより性能に大きな差が生じることがある．なお非ゼロ要素の配置は掛け合わせる対象であるベクトルのアクセスパターンについても影響し，キャッシュヒット率が変わることで性能に影響する可能性があるが，一般的にはそれほど大きな影響とならないため本稿では議論の対象としない．

バランスの悪い行列に対する SpMV の実行時間を短くするためには，行ごとに計算を行う以外の SpMV 実装，すなわち 1 行を分割して計算するような SpMV 実装を用いる必要がある．既存の研究においては，バランスの悪い行列に対する SpMV 実装の高速化については CRS 形式以外の疎行列格納形式，例えば ELL 形式に対するブロック化やスライス化などの実装において注目されているようである [1], [2]．一方で我々は CRS 形式をベースとしたバランスの悪い行列に対する SpMV 実装の高速化に着目しており，文献 [6] では OpenMP のスケジューリング設定を変えることで非ゼロ要素の配置による性能への影響を軽減し，バランスの悪い CRS 形式の疎行列に対する SpMV 実行時間が短縮される例を示している．また文献 [7] においてはバランスの悪い行列を主な対象とした SpMV の実装である BSS 法の提案を行っている．BSS 法は CRS 形式をベースとしながら行単位ではなく固定長のセグメント単位での計算を行っており，1 行を分割して計算するような SpMV 実装となっている．

2.3 CUDA を用いた SpMV の並列化

GPU(CUDA) を用いた SpMV の並列化実装に関する研究も数多く実施されている [3], [4], [5], [8]. GPU は世代更新にともない新たな機能が搭載されることや新たな最適化のパラメタが明らかになる (利用可能になる) ことがあるため，それにあわせて新たな実装方法が提案される傾向にある．

GPU のアーキテクチャおよび対応する実装と最適化の特徴として，並列化を 2 階層で考える点があげられる．すなわち，単一スケジューラにより制御される複数の計算ユニットからなる WARP と，複数の WARP を束ねた Streaming Multiprocessor(以下 SM) である．CRS 形式の疎行列に対する GPU を用いた一般的な並列化手法の例としては，行単位の計算を WARP に割り当てるという手法があげられる．この手法に対応する CUDA プログラム (ソースコード) 例は**図 3** に示す通りである．例示したプログラムは Kepler アーキテクチャの GPU において妥当な性能の得られるコードの例であるが，WARP の制御単位が 32 であることから，行あたりの非ゼロ要素数が少ない行列においては単一の WARP に複数行の計算を行わせた方が高い性能が得られることがある．逆に行あたりの非ゼロ要素数が多い場合には，複数 WARP や複数 SM により計算した方が高い性能となりやすい．そのため OpenMP と同様にバランスの悪い行列において全ての行に対して最適な計算の割り当てを行うことは困難であり，高い性能を得るためには適切に計算を割り当てるような工夫が必要である．

GPU 向けに動的な並列実行機構を用いて SpMV を実施した例としては，Arash らによる手法 [8] があげられる．Arash らの提案する Adaptive-CSR 法においては，非ゼロ要素数の多い行の計算において動的な並列実行機構を活用している旨が述べられている．Adaptive-CSR 法は行あたり非ゼロ要素数をもとに行をまとめ，行あたり非ゼロ要素数にあわせて SpMV アルゴリズムを切り替えるといった手法を採り入れている．しかし多くの行列における性能や詳細な性能最適化パラメタの検討については詳細に述べられていない部分もあり，さらなる評価やパラメタ最適化の余地があると考えられる．

```
int tid, tid2, nwarps;
int iBegin, iEnd, iStep;
tid = threadIdx.x;
nwarps = blockDim.x / 32;
tid2 = tid % 32;
iBegin = blockIdx.x * nwarps + threadIdx.x / 32;
iEnd = nline;
iStep = gridDim.x * nwarps;
for(i=iBegin; i<iEnd; i+=iStep){
  double tmp = 0.0;
  for(j=irp[i]+tid2; j<irp[i+1]; j+=NX){
    tmp += value[j] * vector[icol[j]];
  }
  int k = 1;
  while(k<=16){
    tmp += __hiloint2double(
      __shfl_xor(__double2hiint(tmp),k,32 ),
      __shfl_xor( __double2loint(tmp),k,32 ) );
    k <<= 1;
  }
  if(tid2 == 0)answer[i] = tmp;
}
```

図 3 CRS 形式の疎行列に対する SpMV の CUDA を用いた並列化実装例

3. 実行環境と対象行列

本章では本稿で用いる実行環境と対象行列について述べる．

本稿で実施する性能評価の実行環境を**表 1** に示す．CPU(OpenMP) に関する性能評価については，CPU1 として Xeon E5-2680 v2 (IvyBridge-EP)[9] 1 ソケット 10 物理コア，CPU2 として東京大学情報基盤センターにて運用されているスーパーコンピュータシステムである Oakleaf-FX に搭載された SPARC64IXfx[10] の 1 ソケット 16 コア (=1 ノード) を用いる．MIC(OpenMP) に関する性能評価については，CPU1 を搭載した PC に搭載された Xeon Phi 5110P (Knights Corner)[11] 1 カード 60 コア (最大 240 スレッド) を用いる．MIC において使用するスレッド数は常に 240 とし，アフィニティ設定については balanced を指定する．GPU(CUDA) に関する性能評価については，Tesla K40 (Kepler)[12]1 枚を用いて実施する．コンパイラとしてはそれぞれ Intel コンパイラ icc 15.0.1, 富士通コンパイラ fccpx Fujitsu C/C++ Compiler Driver Version 1.2.1, CUDA 6.5 (Cuda compilation tools, release 6.5, V6.5.12

に gcc version 4.4.7 を組み合わせたもの) を用いる．

性能評価対象とする疎行列については，大きく 3 種類の行列群を用意した．第 1 群は動的な並列実行機構による性能向上が期待できないバランスの良い行列，第 2 群は動的な並列実行機構による性能向上が期待できるバランスの悪い行列，第 3 群はより一般的と思われる行列である．

第 1 群としてはシンプルな 2 次元の 5 点ステンシル行列および 3 次元の 7 点ステンシル行列を用意した．これらの行列は非常にバランスが良い行列であり単純な行単位の並列化に適しているため，動的な並列実行機構による性能向上は期待できない．主にオーバーヘッドの確認に利用する．以降，この行列群を**ステンシル行列**と呼ぶ．

第 2 群としては，対角部分および第 1 行の全ての列のみに非ゼロ要素を含む特別な行列を用意した．この形状の行列は行を分断するような実装でなくては並列化により高速化させにくい行列であり，本稿における動的な並列実行機構による性能向上が期待できる行列である．以降，この行列群を**人工行列**と呼ぶ．

第 3 群としては，Florida Sparse Matrix Collection[13] に含まれる幾つかの行列を用意した．対象となる行列の形状に関する情報を本稿末尾の**表 2** に示す．この行列コレクションには様々な形状の行列が含まれているが，動的な並列実行機構による性能向上が期待できる行列はある程度バランスの悪い行列であることを踏まえて，平均非ゼロ要素数の 2 倍以上の非ゼロ要素数を持つ行が存在する 52 行列のみを用いた．以降，これらの行列を**フロリダ行列**と呼ぶ．

なお，全ての疎行列の非ゼロ要素のデータ型は 64bit 倍精度浮動小数点形式 (double 型配列) とする．また実行時間の比較においては 10 回以上かつ合計 1 秒以上連続実行した際の平均実行時間を使用する．

4. OpenMP 向けの実装と性能評価

4.1 OpenMP を用いた動的な並列実行機構

OpenMP は 1997 年にバージョン 1.0 の API 仕様が公開された後に何度かの更新や拡張が行われており，最新のバージョンは 2013 年に公開されたバージョン 4.0 である．本稿で利用する動的な並列実行機構は 2008 年に公開されたバージョン 3.0 に含まれる機能 [14] であり，**task 構文**を用いて並列実行可能なタスクリージョンを規定するものである．

OpenMP のタスク並列実行機構を用いた簡単なプログラムの例を**図 4** に示す．図 4 中では parallel 指示子により並列ブロックが定義され，その中にある task 構文にてタスクの生成が行われる．図中にて task 構文の直後にあるのは hoge という関数であるため，各タスクの処理内容は hoge 関数ということになる．全てのスレッドは taskwait 構文 (明示されていない場合には並列ブロックの終了時) にて全てのタスクが終了するのを待つ．未割り当てのタスクと，タスクなどの処理が割り当てられていないスレッドが存在する場合には，スレッドに対してタスクが割り当てられて実行される．

```
#pragma omp parallel
{
  int i;
#pragma omp parallel for
  for(i=0; i<n; i++){
#pragma omp task
    hoge();
  }
#pragma omp taskwait
}
```

図 4 OpenMP の task 構文を用いた並列化の例

このように OpenMP の動的な並列実行機構はループの並列化にとらわれない柔軟な並列処理を行える機構である．しかし，多数の計算コアを持つマルチコア CPU や，さらに多くの計算コアを持つメニーコアプロセッサにおいては，タスクの生成や割り当てといった管理処理によりある程度のオーバーヘッドが生じて性能向上が得られない可能性も想定される．また，動的な並列実行機構を用いる場合のデメリットとしてキャッシュヒット率の低下が考えられる．これは SpMV 自体のキャッシュよりもむしろ周囲の計算との兼ね合いに関する問題である．SpMV 自体を繰り返し実行する場合や，直前もしくは直後にも行列に対してなんらかの処理をする並列ブロックがある場合において，静的な並列実行機構では行列の部分がどのスレッド (計算コア) に割り当てられるかが固定化されるために前後の計算も含めて低階層のキャッシュにデータが残ることが期待される (キャッシュヒットしやすいようなプログラムを書きやすい) 一方，動的な並列実行機構では行列の部分がどのスレッド (計算コア) に割り当てられるかが固定化されないために前後の計算も含めると低階層のキャッシュからデータが追い出されやすくなる (キャッシュヒットしやすいようなプログラムを書きにくい) 可能性がある．そのため，静的な並列実行機構と比較してキャッシュヒット率の低下および実行時間の増加を招く可能性がある．

4.2 実装

OpenMP のタスク並列実行機構は，商用か非商用かを問わず，多くのコンパイラによってサポートされている．しかしながら，それを活用した性能の報告数は多くないため，オーバーヘッドの確認およびオーバーヘッドを小さく抑える実装技術について検討や評価を行う必要がある．そのため，本稿では SpMV の高速な実行のアイディアに基づき，幾つかの SpMV 実装を用意して性能を比較する．

そもそも動的な並列実行機構 (タスク並列処理) により SpMV の性能が向上すると考えられる理由は，特定の数行に多数の非ゼロ要素が存在し単純な行ごとの並列計算では

表 1 実験環境

	CPU1	CPU2	MIC	GPU
名称	Xeon E5-2680 v2 (IvyBridge-EP)	SPARC64 IXfx	Xeon Phi 5110P (Knights Corner)	Tesla K40 (Kepler)
動作周波数	2.8 GHz	1.848 GHz	1.053 GHz	745 MHz
コア数（有効スレッド数）	10 (20*1)	16	60 (240)	2880
メモリ種別	DDR3	DDR3	GDDR5	GDDR5
キャッシュ構成	L1 32KB+32KB/core L2 256KB/core L3 25MB/socket	L1 32KB/core L2 12MB/socket	L1 32KB/core L2 512KB/core	OnChip 64KB/SMX R/O 48KB/SMX L2 1536KB/card
理論演算性能	224 GFLOPS	236.5 GFLOPS	1.01 TFLOPS	1.43 TFLOPS
理論メモリ性能	59.7 GB/s	85 GB/s	320 GB/s	288 GB/s
TDP	130W	110W	225W	235W

*1: ただし本稿の実験環境では HT 機能を有効化していない.

```
int threshold = 行あたり非ゼロ要素数が非常に多いと
判断される何らかの閾値;
#pragma omp parallel
{
#pragma omp for
  for(i=0; i<nline; i++){
    int len = 行あたり非ゼロ要素数;
    if(len < threshold){
      1行分のSpMV計算を1スレッドで行う
    }else{
      1行分のSpMV計算の複数のタスクに分解し、
      それぞれ別のスレッドで計算を行う
    }
  }
}
```

図 5 バランスの悪い行列に対して有効な OpenMP の task 構文を用いた SpMV の実装イメージ

一部のスレッドの計算量が多くなりすぎるような場合に，該当行の計算 (の一部) をタスクとして分離して他のスレッドにも手伝わせられるためである．この動作を擬似コードを用いて示すと，例えば図 **5** のようなコードとなる．

ここで，具体的な実装においてはいくつかの選択肢や最適化パラメタが考えられる．

まずは，行あたりの非ゼロ要素数が多いか否かの判定基準である．今回はバランスが悪い行列を想定していることから，対象行の非ゼロ要素数が n 個以上であれば多いと判断する，というような基準よりも，行あたり平均非ゼロ要素数の n 倍以上の非ゼロ要素がある場合は多いと判断する，といった基準が妥当であると考えられる．基準となる数が小さすぎる場合には過剰な数のタスクが生成されたりループ長が短くなるといった性能低下要因が発生することが予想されるため，具体的に何倍程度が妥当なのかは調査が必要である．

実装の方法についても，以下のようにいくつかの選択肢があげられる．例えば 1 行分の SpMV 計算をタスクに分解する際には，いくつのタスクに分解するかや，1 タスクあたりいくつの非ゼロ要素を担当するかは調整可能であり，これらの数量は性能に影響があると考えられる．また行あたり非ゼロ要素数が閾値よりも少ない場合にもタスク化して計算を行えば，スレッド間の負荷均衡化がさらに促進される可能性がある．しかし，基本的に細かいスレッドを多数生成した方が負荷の均衡化が促進されるという考え方も，大量のスレッドは OS や OpenMP 処理系 (タスクスケジューラ) に負担がかかり性能低下を引き起こすという考え方もあるため，ここでは実際にいくつかの実装を行って性能を比較する．

また，図 5 の実装は行ごとのループ内に条件分岐が入っている．行あたり 1 回の分岐であるため性能に非常に大きな影響を及ぼすものではないと思われるが，対象とする行列が非常に小さい (総非ゼロ要素数が少ない) ような場合には有意な性能低下を引き起こすことも考えられる．これについては性能評価の際に考慮する．

各環境における主なコンパイルオプションは以下の通りとした．

- CPU1 : -O3 -openmp -mavx -xAVX -restrict
- CPU2 : -Kfast,openmp,prefetch_indirect,simd=2,restp=arg
- MIC : -O3 -openmp -mmic -restrict

4.3 性能評価

性能比較の対象として，

(1) 行列全体をスレッド数で均等に分割した実装 (スケジューリング設定は行わない)
(2) static,1 のスケジューリングを指定して 1 行ずつ順番に割り当てる実装
(3) dynamic,1 のスケジューリングを指定して 1 行ずつ動的に割り当てる実装
(4) 第 1 行のみ行内計算を OpenMP 並列化 (reduction 指示文を使う) し，さらに残りの行を OpenMP 並列化した実装 (いずれの並列化ブロックについてもスケジューリング設定は行わない)
(5) 全ての行をタスクとして計算する実装 (parallel for 内

```
int i, j;
double tmp = (double)0.0;
#pragma omp parallel for reduction(+:tmp)
for(j=irp[0]; j<irp[1]; j++){
 tmp += value[j] * vector[icol[j]];
}
answer[0] = tmp;

#pragma omp parallel for
for(i=1; i<nline; i++){
  int j;
  double tmp = (double)0.0;
  for(j=irp[i]; j<irp[i+1]; j++){
    tmp += value[j] * vector[icol[j]];
  }
  answer[i] = tmp;
}
```

図 6 OpenMP を用いた SpMV の実装 (実装 4: 「人工行列」向けの実装)

```
#pragma omp parallel
{
  int i;
#pragma omp for
  for(i=0; i<nline; i++){
#pragma omp task
    {
      int j;
      double tmp = (double)0.0;
      for(j=irp[i]; j<irp[i+1]; j++){
        tmp += value[j] * vector[icol[j]];
      }
      answer[i] = tmp;
    }
  }
}
```

図 7 OpenMP を用いた SpMV の実装 (実装 5: task 構文を用いて全行をタスク化する例)

の各行の計算に parallel task を適用する)

以上 5 つの SpMV カーネルを用意した (**図 6**, **図 7**)．実装 1,2,3 は図 2 およびその指示文に schedule 指示子を追加したものである．実装 1 は一般的な実装の一例である．実装 2 や 3 は行列の形状によっては実装 1 よりも良い性能が得られることもあるが，著者らの経験上ではあまり良い実装ではなく，参考のための実装である．実装 4 は人工行列の理想的な最適化の例として用意したものであり，一般的な実装ではない．実装 5 は実装 1,2,3 と比べることでタスク処理のオーバーヘッドがどの程度であるかを確認するためのものであり，性能改善が期待できるものではない．

つづいて，実装のアイディアを基に task 構文を用いた実装を行った．OpenMP 指示文の入れ方などにより細かな違いのある計算カーネルを数パターン用意し，これらにたいしてサイズの異なる数個の人工行列を用いて性能を確認し，タスク処理のオーバーヘッドが目立ちすぎるなどの理由により明らかに低速となる実装を除外した．その結果，性能向上が得られる可能性のある実装として以下の実装群が残った．

(6) 対象とする行が平均の A(A=2,3,4,...) 倍以上の非ゼロ要素数を持つ場合に 1 行を B(B=要素数/コア数, 要素数/コア数の半分) ごとのタスクに分割する実装

なお，さらに最後に残った端数部分をタスク化するかしないかにより多少の性能差が生じるケースがあったため，これについても実験対象として残した．これら実装の擬似コードを**図 8** に示す．

```
#pragma omp parallel
{
  int i;
#pragma omp for
  for(i=0; i<nline; i++){
    int len = irp[i+1] - irp[i];
    if(len < α){// 行あたり要素数の確認
      1行全体のSpMV計算
      answer[i] = 計算結果;
    }else{
      int done = 0, offset = 0;
      answer[i] = 0.0;
      while(done+β<len){
        offset = done;
#pragma omp task firstprivate(offset, i)
        {
          行内のSpMV計算、β要素分ずつ
#pragma omp atomic
          answer[i] += 部分結果;
        }
        done += β;     実装のバリエーションによっては
      }                最後のtask指示文は書かない
#pragma omp task firstprivate(i)
      {
        行内のSpMV計算、残り要素分
#pragma omp atomic
        answer[i] += 部分結果;
      }
    }
  }                  α : 本文中の「A」に基づく数値を設定する
}                    β : 本文中の「B」に基づく数値を設定する
```

図 8 OpenMP を用いた SpMV の実装 (実装 6: task 構文を用いた妥当と思われる実装)

比較基準である 5 カーネルおよび実装 6 群の 4 バリエーション，合計 9 の実装について，ステンシル行列と人工行列を対象として性能を比較した．なお，ステンシル行列はバランスの良い行列であり，人工行列は 1 行目のみ非ゼロ要素数が多いことから，ここでタスク処理を行うか否かの閾値を変えて測定する意味は無い．実装 1 の実行時間に対する他の実装の相対的な実行時間を**図 9** に示す．なお，今回は性能の悪いケースについてはそれが十分に遅いことが

わかれば十分であるため，グラフの上端を実装 1 と比べて 3 倍で打ち切っている．また凡例中の A,X,B については以下を意味する：

A2, A3, A4, A5 行を分解するか否かの閾値を平均非ゼロ要素数の何倍とするか．2 は 2 倍以上，3 は 3 倍以上，4 は 4 倍以上, 5 は 5 倍以上．

B1, B2 1 行をいくつのタスクに分解するか．B1 はスレッド数分に，B2 はスレッド数の半分に分解．

C1, C2 「残り要素分」をタスクとして計算するか，そのまま計算するか．C1 はタスクにする，C2 はしない．

各実験環境ごとに性能の傾向を確認する．まず全体に共通して確認できることとして，実装 2,3,5 の実行時間は実装 1 より明らかに長い．特に全行をタスクとして計算する実装 5 の実行時間が長いことから，タスク構文を用いた動的な並列実行機構の動作にはある程度大きなオーバーヘッドがあると考えた方が良い．また実装 4 は人工行列において高速であるのはもちろん，ステンシル行列においても総非ゼロ要素数が多い場合には実装 1 と同程度の性能を得られているが，これはプログラムの構造的には妥当な結果である．

つづいて CPU1 における結果について確認する．総非ゼロ要素数の少ないステンシル計算において，タスク構文を用いた実装である実装 6 群はいずれも実装 1 の 1.5 倍程度の実行時間を要しているのに対して，総非ゼロ要素数の多いステンシル計算では同程度実行時間となった．総非ゼロ要素数の少ない場合に実行時間比が大きくなっているのは，実装の説明で述べたとおり，ループ内に分岐が入ったことによるオーバーヘッドによるものであり，総非ゼロ要素数が多い場合には 1 ループあたりの実行時間や全体の実行時間全体が長くなって目立たなくなったものと考えられる．人工行列においては，総非ゼロ要素数が少ない場合には実装 6 群の実行時間は実装 1 と比較して長いものの，総非ゼロ要素数が多い場合には半分程度の実行時間比へと高速化できており，task 構文を用いた実装の効果がよくわかる結果となった．

さらに CPU2 における結果について確認する．CPU2 の結果は CPU1 の結果と比べて，総非ゼロ要素数の少ないステンシル行列における実装 6 群の性能低下具合 (実装 1 に比べた実行時間比の増加具合) が小さく，ほぼ同様の実行時間であった．これはループ内に多少の分岐があっても影響しない，もしくは分岐の方向が常に一定であるため分岐予測等の仕組みにより性能低下が抑えられた可能性が考えられる．ただし，この結果を見ただけでは，実装 1 が遅いのか実装 6 群が速いのかの判別はできない．また，総非ゼロ要素数が少ない人工行列における実装 6 群の実行時間比は CPU1 よりも大きく，総非ゼロ要素数が多い人工行列においては逆に CPU1 よりも小さいという結果も確認できる．

最後に MIC における結果であるが，人工行列において実装 6 群の性能が優位となる総非ゼロ要素数のの目安が大きくなった以外には CPU2 と大差ない結果であった．この結果から，MIC についてはループ内に分岐処理が入ったことによる性能低下は小さいものの，task 構文を用いた動的な並列実行機構のオーバーヘッドは大きいと考えられる．

実装 6 群同士の性能についてはあまり大きな差は無かったが，実装 6 群が良い性能を得られていない際には B2 群が，得られている際には B1 群がやや優位な傾向が見られる．B1 群はコア数分，B2 群はコア数の半分分のタスクを生成する仕組みであることを考慮すると，実装 6 群が良い性能を得にくいような場合にはコア数が増えることでより大きな性能ペナルティ (オーバーヘッド) が生じ，実装 6 群が良い性能を得られるような場合にはコア数が増えるとより加速している，という状況であるように見える．どの程度のタスクを生成すれば常に最大の性能が得られるかについてはさらなる評価や検証が必要でああろう．

次に，フロリダ行列を用いた性能評価結果について確認する．図 9 と同様，**図 10** に実装 1 に対する実装 6 群の各実装の実行時間比を示す．各グラフとも右側ほど総非ゼロ要素数の多い行列が対象となっている．今回はタスクにより行を分解する閾値も性能に影響を及ぼすと考えられるため，平均比ゼロ要素数の 2 倍,3 倍,4 倍,5 倍のパターンを用意した．これらの値は実装バリエーションの A=2,3,4,5 に該当する．

全体の結果を見てみると，CPU1 と MIC においてはほとんどの行列で実行時間をあまり短くできていない．実装 1 と比べて 5%以上の性能向上が確認できた行列は CPU1 で 2 行列，MIC でも 11 行列と少なかった．もっとも多く実行時間を削減できた行列は，CPU1 では thread(16.1%, 設定 A5, B2, C1), MIC では shipsec1(26.5%, 設定 A4, B2, C2) であった．

一方で CPU2 においてはグラフ左側に位置する小規模な行列に対しては CPU1 や MI と同様にほとんど性能向上が得られなかった一方で，グラフ右側に位置する大規模な行列に対しては多くの行列において性能向上が得られた．実装 1 と比べて 5%以上の性能向上が確認できた行列は 20 あり，行列 bmwcra_1(設定 A5, B1, C2) において最大で 42.3%の実行時間を短縮できた．

実装のバリエーションとして設けた A,B,C の 3 パラメタについては，パラメタ A は全体的に 5 が一番良い，つまり平均非ゼロ要素数との差が大きな場合に良い性能が得やすいことを意味しており，タスク処理のオーバーヘッドが無視できない程度に大きな状態であると考えれば妥当である．パラメタ B は 1 よりも 2 の方が良いケースが散見された．つまり行ごとに作成するタスクは小さく多くよりも大きく少なくの方が良いことを意味する．今回の実装は閾値を超えた各行がタスクを生成するため，多くの行が閾値を超えるとその分だけ多くのタスクが作成されることにな

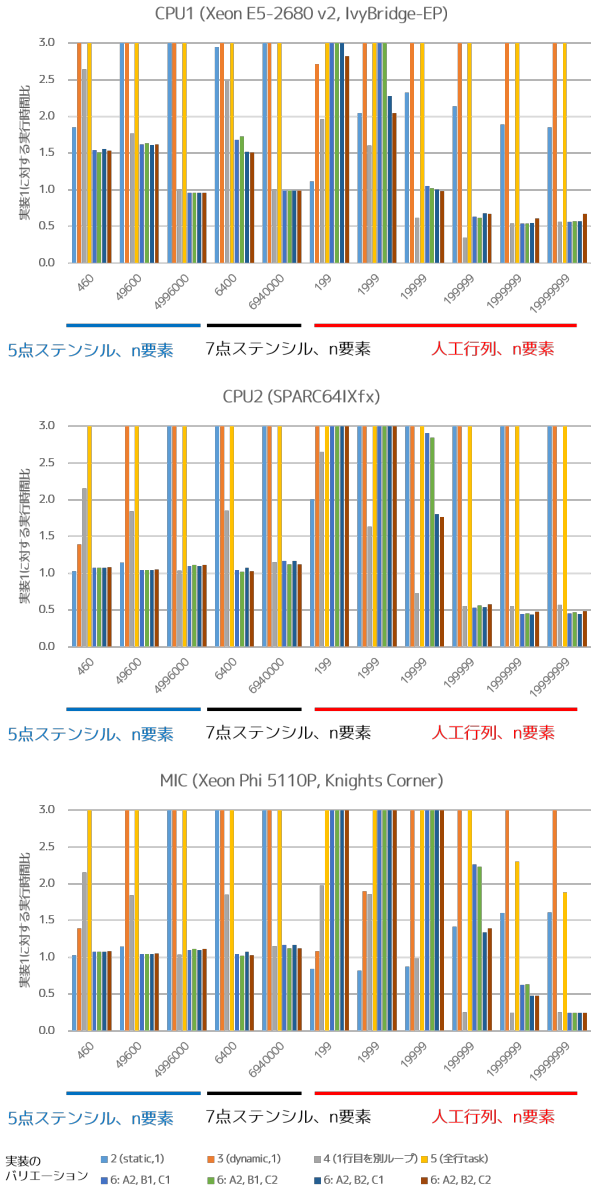

図 9　OpenMP を用いた SpMV の実行時間 (横軸の数値は n 要素の n に対応する)

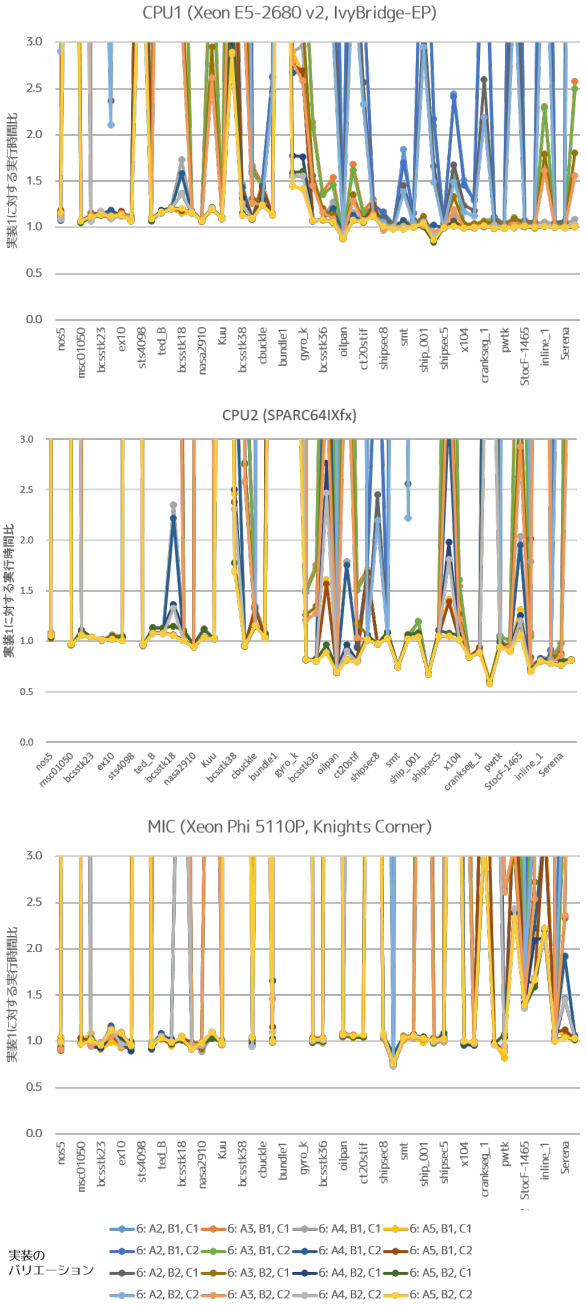

図 10　OpenMP を用いた SpMV の実行時間 (フロリダ行列)

る．過剰に多くのタスクを生成するとオーバーヘッドが上昇し性能が低下するため，タスク生成数を半分にした B2 の方が良い結果が得られたものと考えられる．パラメタ C はあまり影響が確認できなかった．

さらなる性能向上のためには各種の最適化パラメタの精査が必要であると考えられる．具体的には，タスク化を行うか否かの判断基準の最適化や，タスク化を行う際に何要素ずつのタスクへと分割するかについてはさらに最適化を行う余地があると考えられる．

5. CUDA 向けの実装と性能評価

5.1 CUDA を用いた動的な並列実行機構

CUDA は 2007 年初頭に一般公開が開始された後，主に GPU のアーキテクチャ更新とともにバージョンアップを行い，現在ではバージョン 6.5 が一般公開されている．本稿で利用する動的な並列実行機構は Kepler アーキテクチャから対応された dynamic parallelism 機構である [15]．この機構は GPU カーネルから他の GPU カーネルを実行できる機構である．従来の CUDA においては GPU カーネルを実行するためには一度 CPU に制御を戻す必要があった．そのため，GPU 上で計算を始めてしまったあとで GPU カーネルに割り当てる計算資源の量を変えることはできず，GPU カーネル全体の終了を待ったうえでホスト CPU から改めて GPU カーネルを実行する必要があった．dynamic parallelism 機構を用いることでこの制限が緩和されるため，従来よりも効率よく GPU を利用できる

可能性が高まる．

5.2 実装

CUDA を用いた CRS 形式の疎行列に対する SpMV 実装については，2.3 節で述べたように既に多くの研究事例が存在する．しかし，任意の形状 (非ゼロ要素の配置) の疎行列に対して常に最大の性能を得る実装は，例えバランスの悪い行列が対象ではないとしても，CPU と比べて少し難しい．なぜならば，1 行あたりいくつの WARP を割り当てるか，もしくは 1WARP につき何行を割り当てるか，これらの最適値が対象となる行列により異なるためである．今回は様々な最適化パラメタを詳細に比較するよりも dynamic parallelism の効果を確認することおよび OpenMP による実装と比較することを主眼に置き，一般的な最適化手法を適用された GPU カーネル (以下**静的カーネル**) と，それを基に dynammic parallelism を適用されたカーネル (**動的カーネル**) を 1 組作成して性能を比較した．

静的カーネルとしては，**図 11** に示すような GPU カーネルを実装した．WARP や ThreadBlock の数などのパラメタについては以下のように考えて作成した．1WARP(32threads) が 1 行を計算する．1ThreadBlock あたりの Thread 数は 128 に固定する．すなわち 1SM あたり 4 行を担当する計算となる．grid のサイズ (ThreadBlock 数) を行列の行数にあわせた最低限の数とすることで GPU カーネル内のループを削減している．WARP シャッフル命令を用いた集約演算の実現や関数の引数に対する restrict キーワードの追加といった一般的な最適化手法も適用している．

1WARP が 1 行を担当することや 1ThreadBlock が 128Thread から構成されるといった点については CUDA 向け最適化 SpMV プログラミングにおいて妥当な設定の範囲であると考えられるが，1WARP が 1 行を計算するため行あたりの非ゼロ要素数が 32 未満の場合には計算ユニットが遊ぶことになり，また逆に非ゼロ要素数が大きな場合でも 1WARP のみで計算をせねばならない．そのため様々な疎行列に対して十分な性能を引き出すには，これらの値を適切に変更する必要がある．本稿の動的な並列実行機構には行あたり非ゼロ要素数が大きな行が混在した場合の性能改善が期待される．

動的カーネルとしては，**図 12** に示すような GPU カーネルを実装した．基本的な演算処理は静的カーネルと同様であるが，OpenMP の実装 6 群と同様に，行あたり非ゼロ要素数がある程度大きな場合に GPU カーネルを動的に実行するようにした．この際に実行される GPU カーネルは，行あたり非ゼロ要素数の大きな行をより効率よく計算できるように 1ThreadBlock * 128Thread 全てを用いて 1 行を計算する作りとした．なお，動的に実行する GPU カーネルを 4ThreadBlock * 128Thread に変更した場合の性能についても測定したが，1ThreadBlock * 128Thread の場合と有意な性能差がなかったため省略する．さらに，閾値を超えたとしてもあまり非ゼロ要素数の多くない行を別カーネルにて実行するのは非効率であると考え，128 要素未満の場合には強制的に静的カーネル同様に処理する仕組みを導入している ($||(len < 128)$ の部分が該当)．

主なコンパイルオプションは以下の通りとした．

-O3 -gencode arch=compute_35,code="sm_35,compute_35" -rdc=true

```
__global__ void spmvfunc_static(……){
  int i, j, tid, tid2;
  int row = blockIdx.x * 4 + threadIdx.x/32;
  tid = threadIdx.x;
  tid2 = threadIdx.x % 32;
  if(row < nline){
    double tmp = 0.0;
    for(j=irp[row]+tid2; j<irp[row+1]; j+=32){
      tmp += value[j] * vector[icol[j]];
    }
    int k = 1;
    while(k<=16){
      tmp += __hiloint2double(
          __shfl_xor(__double2hiint(tmp),k,32),
          __shfl_xor(__double2loint(tmp),k,32));
      k <<= 1;
    }
    if(tid2 == 0)answer[row] = tmp;
  }
}
```

図 11 CUDA を用いた SpMV の実装 (静的カーネル)

```
__global__ void spmvfunc_inner(……){
  // 静的カーネルとほぼ同様の計算
  // ただし128Thread全てを使って1行を計算する
}

__global__ void spmvfunc_dynamic(……){
  int i, j, tid, tid2;
  int row = blockIdx.x * 4 + threadIdx.x/32;
  tid = threadIdx.x;
  tid2 = threadIdx.x % 32;
  int nAvg = irp[nline]/nline;
  if(row < nline){
    int len = irp[row+1] - irp[row];
    if((len < nAvg * factor) || (len < 128)){
      // 静的カーネルと同様の計算
    }else{
      if(tid2==0){
        spmvfunc_inner<<<1,128>>>(.....);
      }
    }
  }
}
```

図 12 CUDA を用いた SpMV の実装 (動的カーネル)

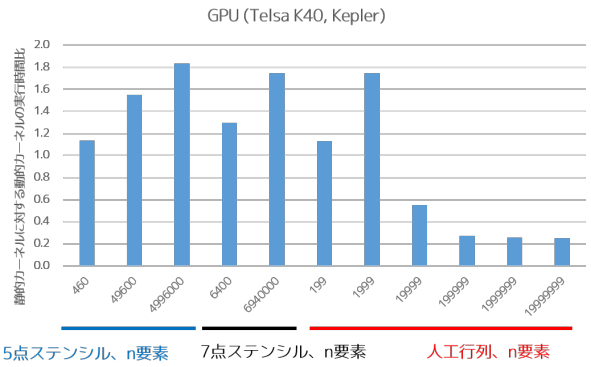

図 13　CUDA を用いた SpMV の実行結果 1(ステンシル行列，人工行列)

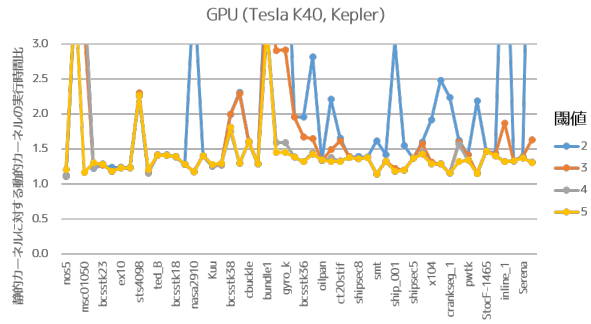

図 14　CUDA を用いた SpMV の実行結果 2(フロリダ行列)

5.3　性能評価

実装した 2 つのカーネルを用いて，CPU や MIC と同様に各行列についての実行時間を測定して比較した．実行時間の測定範囲には CPU-GPU 間のデータ転送は含まず，CPU や MIC と同様に 10 回以上かつ 1 秒以上連続実行して平均時間を取得した．実行結果を**図 13** および**図 14** に示す．

まずステンシル行列における結果を確認する．動的カーネルの実行時間が静的カーネルに比べて長いのは CPU や MIC と同様の傾向であるが，CPU や MIC と異なり総非ゼロ要素数が少ない場合に性能差が小さく，総非ゼロ要素数が多い場合に性能差が大きいという結果が得られた．理由としては，行あたりの非ゼロ要素数が増えずに行数のみが増えた形となるため差が広がった可能性が考えられる．CPU や MIC と傾向が異なる点についてはさらなる評価検証を検討したいが，一般的に GPU は命令スケジューリング機構等の都合により CPU や MIC よりも分岐処理に弱い，分岐無く連続して実行できる処理に対して特に高性能である傾向があり，今回得られた結果もその傾向に沿ったものではある．

続いて人工行列における性能について確認する．GPU においては特に総非ゼロ要素数が多い際において，CPU や MIC よりも動的カーネルの優位性が際立つ結果となった．総非ゼロ要素数が少ない際の性能の傾向がやや不可解にも見えるが，単純に実行時間の伸び具合が静的カーネルと動的カーネルで同一ではなかったためであると考えている．

さらにフロリダ行列における性能について確認する．動的カーネルの実行時間は全てのケースにおいて静的カーネルより長く，動的な並列実行機構による性能改善は行えなかった．

以上のように，dynamic parallelism による性能向上は人工行列のみでしか得られておらず，今回実装と実験を行った範囲においては限定的な効果であった．より高い性能を得るためには，適切なパラメタの検討が必要であると考えられる．例えば GPU カーネル起動時のグリッドサイズやブロックサイズはホスト実行時と動的カーネル起動時の両方を最適化する必要があるだろう．もちろん OpenMP のタスク並列化機構と同様に dynamic parallelism を用いるか否かの閾値などについても検討の余地がある．

6.　おわりに

本稿ではバランスの悪い行列 (行あたりの非ゼロ要素数が均等に近くない疎行列) に対する疎行列ベクトル積の高速化を目指して動的な並列実行機構を用いた SpMV の実装と評価を行った．CPU と MIC に対しては，OpenMP の task 構文を用いたタスク並列処理による実装を行った．実験の結果，意図的に作られた非常にバランスの悪い行列 (人工行列) に対しては大きな速度向上を得た．しかしより一般的な行列 (フロリダ行列) においては，SPARC64IXfx においてはいくつかの行列において性能が向上し動的な並列実行機構の有効性が確認できた一方，Xeon や Xeon Phi においてはあまり効果が得られなかった．また GPU に対しては，CUDA の dynamic parallelism 機構を用いた実装を行った．実験の結果，人工行列に対しては CPU や MIC よりも大きな速度向上を得た一方で，フロリダ行列については速度向上が得られなかった．

以上のように，動的な並列実行機構を用いた SpMV の実装は大きな性能向上が得られる事例がある一方で，タスク処理のオーバーヘッド等の都合により大きく性能低下するケースも多かった．今後は性能最適化パラメタの精査や，状況に応じた動的な並列実行機構の使い分けについてもさらに検討や評価を行いたい．2 種の CPU と MIC のように同じプログラムでも性能の傾向が違ったものについては，現時点ではハードウェアの性能によるものかシステムソフトウェアの性能によるものかも定かではないが，各計算機環境における様々なアプリケーションの最適化へのヒントとなる点がないかについても検討していきたい．さらに，性能低下の要因としては様々な最適化パラメタの精査が不足していることもあるが，動的な並列実行機構そのもののオーバーヘッドも大きい可能性があり，ハードウェアやシステムソフトウェアの改善を期待したい．また本機構を取り入れたアプリケーションやライブラリの実装などにも取

り組みたいと考えている．

謝辞 日頃より最適化プログラミングについて議論をさせていただいている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の皆様に感謝します．本研究は JSPS 科研費 24300004(実行時自動チューニング機能付き疎行列反復解法ライブラリのエクサスケール化), JST CREST「自動チューニング機構を有するアプリケーション開発・実行環境:ppOpen-HPC」の助成を受けたものです．

参考文献

[1] Ali Pinar and Michael T. Heath: Improving performance of sparse matrix-vector multiplication: SC '99 Proceedings of the 1999 ACM/IEEE conference on Supercomputing (1999).
[2] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel: Optimization of sparse matrix-vector multiplication on emerging multicore platforms. SC '07 Proceedings of the 2007 ACM/IEEE conference on Supercomputing (2007).
[3] Nathan Bell and Michael Garland: Efficient Sparse Matrix-Vector Multiplication on CUDA. NVIDIA Technical Report NVR-2008-004 (2008).
[4] Istvan Reguly, Mike Glies: Efficient sparse matrix-vector multiplication on cache-based GPUs. In: Proc. Innovative Parallel Computing: Foundations and Applications of GPU, Manycore, and Heterogeneous Systems (InPar 2012), pp.1-12 (2012).
[5] Daichi Mukunoki and Daisuke Takahashi: Optimization of Sparse Matrix-vector Multiplication for CRS Format on NVIDIA Kepler Architecture GPUs. Proc. The 13th International Conference on Computational Science and Its Applications (ICCSA 2013), Springer LNCS 7975, pp.211-223 (2013).
[6] Satoshi Ohshima, Takahiro Katagiri, Masaharu Matsumoto: Performance Optimization of SpMV using the CRS format considering OpenMP Scheduling on CPUs and MIC, Auto-Tuning for Multicore and GPU (ATMG) session in IEEE 8th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC-14), University of Aizu, Aizu-Wakamatsu, Japan, September 23-25 (2014). / International conference held in Japan
[7] Takahiro Katagiri, Takao Sakurai, Mitsuyoshi Igai, Satoshi Ohshima, Hisayasu Kuroda, Ken Naono, and Kengo Nakajima: Control Formats for Unsymmetric and Symmetric Sparse Matrix-Vector Multiplications on OpenMP Implementations. Selected Paper of VECPAR 2012, Springer LNCS 7851, pp.236-248 (2013).
[8] Arash Ashari, Naser Sedaghati, John Eisenlohr, Srinivasan Parthasarathy, P. Sadayappan: Fast Sparse Matrix-Vector Multiplication on GPUs for Graph Applications. Inproseedings of SC14 (International Conference for High Performance Computing, Networking, Storage and Analysis), pp.781-792 (2014).
[9] Intel Xeon Processor E5-2680 v2 http://ark.intel.com/ja/products/75277/
[10] FX10 スーパーコンピュータシステム (Oakleaf-FX) website, http://www.cc.u-tokyo.ac.jp/system/fx10/
[11] Intel Xeon Phi Coprocessor 5110P http://ark.intel.com/ja/products/71992
[12] NVIDIA Tesla GPUs (Tesla K40) http://www.nvidia.com/object/tesla-servers.html
[13] T. A. Davis and Y. Hu: The University of Florida Sparse Matrix Collection. ACM Transactions on Mathematical Software, Vol 38, Issue 1, pp 1-25 (2011). http://www.cise.ufl.edu/research/sparse/matrices
[14] Version 3.0 Complete Specifications, http://openmp.org/wp/openmp-specifications/
[15] CUDA Dynamic Parallelism, http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cuda-dynamic-parallelism

表 2　計算対象とする疎行列

行列名	行数	総非ゼロ要素数	min	max	avg	max/avg	A-1	A-2	A-3	A-4	A-4	A-5
nos5	468	5172	7	23	11.05	2.08	171	1	0	0	0	0
bcsstk08	1074	12960	1	339	12.07	28.09	226	21	21	19	19	17
msc01050	1050	26198	5	128	24.95	5.13	402	42	29	10	10	0
bcsstk26	1922	30336	3	33	15.78	2.09	869	6	0	0	0	0
bcsstk23	3134	45178	2	31	14.42	2.15	1449	4	0	0	0	0
ex3	1821	52685	8	62	28.93	2.14	1028	4	0	0	0	0
ex10	2410	54840	6	50	22.76	2.20	1082	180	0	0	0	0
ex10hs	2548	57308	4	50	22.49	2.22	1134	180	0	0	0	0
sts4098	4098	72356	4	784	17.66	44.40	1621	203	53	19	16	14
bcsstk13	2003	83883	5	95	41.88	2.27	806	182	0	0	0	0
ted_B	10605	144579	1	49	13.63	3.59	6327	2685	300	0	0	0
ted_B_unscaled	10605	144579	1	49	13.63	3.59	6327	2685	300	0	0	0
bcsstk18	11948	149090	1	49	12.48	3.93	5604	1011	9	0	0	0
Muu	7102	170134	12	49	23.96	2.05	3622	630	0	0	0	0
nasa2910	2910	174296	16	175	59.90	2.92	1350	228	0	0	0	0
bcsstk25	15439	252241	2	59	16.34	3.61	7924	331	38	0	0	0
Kuu	7102	340200	23	98	47.90	2.05	3622	630	0	0	0	0
gyro_m	17361	340431	4	120	19.61	6.12	5826	870	231	51	27	9
bcsstk38	8032	355460	2	614	44.26	13.87	4219	137	19	4	2	2
bcsstk17	10974	428650	1	150	39.06	3.84	6834	6	6	0	0	0
cbuckle	13681	676515	26	600	49.45	12.13	9145	1	1	1	1	1
Andrews	60000	760154	9	36	12.67	2.84	24888	325	0	0	0	0
bundle1	10581	770811	17	1711	72.85	23.49	1158	423	333	315	315	315
gyro	17361	1021159	12	360	58.82	6.12	5826	870	230	51	27	9
gyro_k	17361	1021159	12	360	58.82	6.12	5826	870	230	51	27	9
msc23052	23052	1142686	8	178	49.57	3.59	15541	254	5	0	0	0
bcsstk36	23052	1143140	8	178	49.59	3.59	15553	255	5	0	0	0
msc10848	10848	1229776	45	723	113.36	6.38	4422	261	27	18	6	6
oilpan	73752	2148558	1	63	29.13	2.16	36280	4630	0	0	0	0
vanbody	47072	2329056	6	231	49.48	4.67	24463	960	44	8	0	0
ct20stif	52329	2600295	2	207	49.69	4.17	28784	302	11	6	0	0
nasasrb	54870	2677324	12	276	48.79	5.66	36245	12	12	12	12	0
shipsec8	114919	3303553	3	100	28.75	3.48	48712	3107	23	0	0	0
shipsec1	140874	3568176	4	68	25.33	2.68	61474	4743	0	0	0	0
smt	25710	3749582	51	414	145.84	2.84	15858	210	0	0	0	0
ship_003	121728	3777036	3	92	31.03	2.97	51370	3649	0	0	0	0
ship_001	34920	3896496	3	362	111.58	3.24	12880	1818	4	0	0	0
thread	29736	4444880	28	306	149.48	2.05	18989	231	0	0	0	0
shipsec5	179860	4598604	2	76	25.57	2.97	80630	4912	0	0	0	0
bmw7st_1	141347	7318399	1	435	51.78	8.40	91970	351	135	45	27	14
x104	108384	8713602	8	270	80.40	3.36	29405	1359	12	0	0	0
m_t1	97578	9753570	48	237	99.96	2.37	35496	1596	0	0	0	0
crankseg_1	52804	10614210	48	2703	201.01	13.45	21031	1762	4	4	4	4
bmwcra_1	148770	10641602	24	351	71.53	4.91	92755	378	348	318	0	0
pwtk	217918	11524432	2	180	52.88	3.40	194990	6	6	0	0	0
crankseg_2	63838	14148858	48	3423	221.64	15.44	27700	2248	16	4	4	4
StocF-1465	1465137	21005389	1	189	14.34	13.18	1058721	1142	296	97	49	33
Fault_639	638802	27245944	1	267	42.65	6.26	478742	281	81	17	5	2
inline_1	503712	36816170	18	843	73.09	11.53	153282	28938	3378	33	18	18
Hook_1498	1498023	59374451	1	93	39.64	2.35	714864	67	0	0	0	0
Serena	1391349	64131971	1	249	46.09	5.40	674572	1710	117	33	6	0
audikw_1	943695	77651847	21	345	82.28	4.19	173673	84399	4113	6	0	0

min：行あたり非ゼロ要素数の最小値
max：行あたり非ゼロ要素数の最大値
avg：行あたり非ゼロ要素数の平均値
A-n (n=1,2,3,4,5)：平均の n 倍以上の非ゼロ要素数を持つ行の数