Vol.2014-HPC-146 No.14
2014/10/3

CUDA-xSYMVの実装と評価

今村 俊幸[1,3,a)]　椋木 大地[1]　山田 進[2,3]　町田 昌彦[2,3]

概要：対称行列ベクトル積 (SYMV) は行列の対称性を利用して要求バンド幅を半減できる演算である．適切な最適化技法を利用することで, 一般行列ベクトル積 (GEMV) よりも 2 倍の性能を示すことが期待される．本研究では, 対称性を利用する際に考慮しなくてはならない複数スレッドによるベクトルデータへの書き込み競合に対して, アトミック演算を用いた mutex の実装を工夫することによりアクセス順制御を実現している．これにより, CUBLAS 等で指摘されている「実行毎に丸め誤差の範囲で演算結果が異なる」という現象を回避できる．また, 既存研究ではスレッドブロック形状が 1 次元であったものを 2 次元に拡張し, 計算コア数を増加させることができるようになった．
本研究のもう一つのポイントは自動チューニング技術 (AT) による最適パラメタ探索により高性能カーネルの構築を実現していることにある．2 次元ブロック化によって広範囲に分布するパラメタ空間から自動で最適パラメタ値を探索し, 少々時間を要するものの最適化された高性能 SYMV を GPU アーキテクチャ毎にビルドすることができる．実際, 最適化された SSYMV(単精度版 SYMV) カーネルが, GeForce GTXTitan Black 上で 211GFLOPS(対最大バンド幅 62.8%) を記録している．さらに, 実数 (単精度や倍精度) 以外の数値フォーマットである複素数 (単精度, 倍精度) ならびに疑似四倍精度 DD(double-double) フォーマットに対しても, 同様のアプローチにより SYMV カーネル (CHEMV, ZHEMV, WSYMV) の実装に成功し, 高い実行性能を確認している．

1. はじめに

対称行列ベクトル積 (SYMV) は対称行列とベクトルの積を計算する数値線形代数計算の中でも重要な位置を占める計算パターンである．例えば, 対称密行列の固有値計算の前処理であるハウスホルダー三重対角化においてランク更新計算と並ぶ高負荷計算部分とされている．

$$y := \alpha A^{\mathrm{UorL}} x + \beta y \quad (A(=A^T) \in \mathrm{R}^{n\times n}, x \in \mathrm{R}^n) \quad (1)$$

BLAS に関しては, 理論的かつ実験的解析から各種の性質が知られているが, 特にメモリバンド幅との性能相関が重要である．SYMV は BLAS でレベル 2 に分類されており, 計算量 $O(N^2)$ に対してデータ量が $O(N^2)$ となるために, 要求メモリバンド幅が $O(1)$ となり一般的にはメモリバンド幅によって性能が律速される．

一般的に, GPU は CPU に比べ高いメモリ転送性能を持っており, GPU での BLAS 実装の代表である CUBLAS[1] や MAGMA[2] などでは CPU よりも高い性能を示す．例えば, Sørensen の実装である GLAS [3], [4], [5] では行列ベクトル積の一般版である GEMV はメモリバンド幅換算で 63.8%(NVIDIA Tesla C2050 で 90GB/s=45GFLOPS) に達する．その他多くの報告がなされている [6], [7], [8]．性能最適化部分にも依存するが, SYMV に関して GPU の広いメモリ帯域を活かした高性能実装の報告が存在する [9], [10], [11].

SYMV など対称行列に限定した処理関数 SYxxx 系では, もともと格納行列が上三角もしくは下三角部分のみを保持する．データ総量削減とともに, アルゴリズムの適切化により主記憶–プロセッサ間データ移動量を GEMV と比較して 1/2 に削減できることが知られている [9], [11]．これは, 所謂要求 Byte/flop が 1/2 になることを意味しており, GEMV に対して性能が 2 倍まで上昇できる可能性を示唆している．これまでに, データの対称性を活かしながらデータ移動量を削減するアルゴリズムの報告は数々あったが, 初期の研究では GEMV に対して 2 倍の性能に迫る実装系はほとんど存在していなかった．

近年, SYMV の実装に対してアトミック演算を使用するアルゴリズムが著者らのものも含めて各種提案されている [7], [11]．実際, CUDA 6.0[12] から CUBLAS のオプショ

[1] 理化学研究所 計算科学研究機構
RIKEN Advanced Institute for Computational Science, Kobe, Hyogo
[2] 日本原子力研究開発機構
Japan Atomic Energy Agency, Kashiwa, Chiba
[3] 科学技術振興機構 CREST
CREST JST, Kawaguchi, Saitama
[a)] imamura.toshiyuki@riken.jp

ンとして KBLAS[7] が採用されるようになり [*1] 高性能化が達成されつつある．一方, CUBLAS のマニュアルには非決定的な演算順序のため「実行毎に丸め誤差の範囲で演算結果が異なる」という指摘がある．本研究では, アトミック演算を利用しつつも演算順序を制御し, 演算結果が一意になる工夫を行い, Byte/flop 値削減と計算精度を担保する．さらに, 既存研究 [13] ではスレッドブロック形状が 1 次元であったものを 2 次元に自然な拡張をする．

本研究のもう一つのポイントは自動チューニング技術 (AT) を使用した最適化である．複雑な GPU コードに埋め込まれた複数のパラメタに対する実行性能最適化問題を AT 技術を用いて, 広範囲に分布するパラメタ空間から自動で最適パラメタ値を探索し, GPU アーキテクチャ毎に性能最適化された SYMV カーネルをビルドすることができる．実際 , 最適化された SYMV は他の CUDABLAS 実装に比べて高い性能を記録する ．

さらに ，本研究ではビルトイン型である実数型フォーマット (double, float) の他にも複素数型フォーマット (cuFloatComplex, cuDoubleComplex) への拡張を行う．また, 高精度計算向けの拡張として, 疑似四倍精度 DD(double-double) フォーマット [14] に対しても , 同様のアルゴリズム (アトミック演算を使用する Atomic Algorithm) を採用した SYMV カーネルの実装を行う．

本報告の目的は, 既存研究の拡張として実施した開発の技術内容の整理, ならびに開発した SYMV カーネルの性能測定結果の速報をまとめることである．

2. CUDA-xSYMV の実装方法

2.1 基本アルゴリズム (Atomic algorithm)

対称行列ベクトル積は行列の対称性を利用して行列要素のアクセス回数を 1/2 削減することができる．図 1 は, 行列 A の格納方式が上三角収納形式の場合の逐次 SYMV アルゴリズムである．

2 重ループの再内ループ内に存在する行列データへのアクセス (`Aij=A(i,j)`) 1 回に対して直後に続く 2 回の積和演算が対応している．したがって, 理想的な状況では演算あたりのメモリ要求量は 1word/4flop となる（一般行列ベクトル積では 1word/2flop)．倍精度計算では 8/4=2B/F, 単精度では 4/4=1B/F と定まる．一般に MV 計算はメモリバンド幅で律速する．低い B/F 値を維持するアルゴリズム実現は高性能化に直結する重要事項である．

2.1.1 2 次元スレッドブロック配置

最も素直な並列化によると, スレッドを 2 次元ないし 1 次元形状に設定し, 2 次元もしくは 1 次元ブロック分割した部分行列計算を各スレッドブロックに割り当てていくことがセオリーである．今回, 著者らの先行研究 [11], [13] で 1 次元スレッドブロックを採用してきたものを 2 次元スレッドブロックに自然に拡張する．本拡張の利点・欠点は以下のようになる．

- 1 次元ブロックでは 1 ブロックあたりのスレッド数が制限されるが, 2 次元化により 1 ブロック中のスレッド数が増加する．
- ベクトル v への書き込みに対して, ブロック間に加え Ty 方向スレッド間の排他制御が必要になる．

図 3 は CUDA プログラミングモデルによる図 1 のスレッド並列化の一例を示すものである．図 1 と 3 は, コメント部分の part one から three までのそれぞれが対応する形で記述している．

2.1.2 排他制御

図 1 と図 3 の part one は図 2 の灰色のパネル内処理の中で対角ブロック (赤線) にかかるまでの区間の処理に相当す

```
! Sequential SYMV kernel algorithm
! Compute y:=alpha*A*x+beta*y
!
   v(1:n)=0; y(1:n)*=beta
   ! part one
   do j=1,n
      t=0
      do i=1,j-1
         Aij=A(i,j)
         v(i)+=Aij*x(j)
         t+=Aij*x(i)
      enddo
      y(j)+=alpha*t
   enddo
   ! part two
   do i=1,n
      y(i)+=alpha*A(i,i)*x(i)
   enddo
   ! part three
   y(1:n)+=alpha*v(1:n)
```

図 1 対称性を考慮した逐次型 SYMV アルゴリズム (行列 A が上三角収納形式の場合)

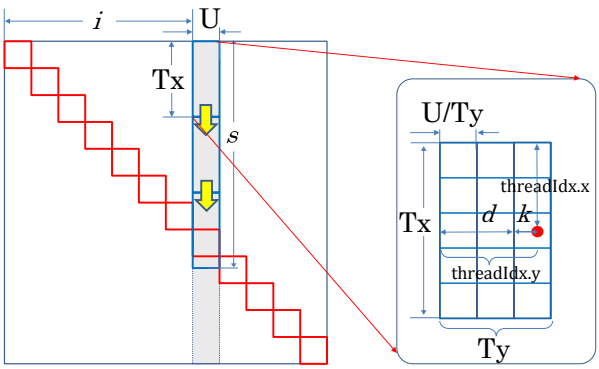

図 2 スレッドブロックとパネルなど各種分割形状パラメタ (Tx, Ty, U) 位置変数 (i, d, k, s) との関係

[*1] デフォルトの状態ではオフにされており，使用するには `cublasSetAtomicsMode` により `CUBLAS_ATOMICS_ALLOWED` を指定する必要がある．

```
kernel symv_preprocess
  j := get_threadID().
  if j < n then
    v(j) := 0, and y(j) *= beta.
  endif
  if j < MAX_blkID then
    ticket(j) := MAX_blkID.
  endif
  if j = 0 then
    atomicExch( &Master_blkID, 0 ).
  endif
endkernel

kernel symv_main <Tx, Ty, U, M>
  define j ≡ J + threadIdx.x.
  thID := get_localID(), and
  blkID := get_blockID().
  d := (U/Ty)*threadIdx.y, i := U*blkID, and
  s := ceil(i − 1, Tx).
  Ticket := ticket.
  yreg[0] := ... := yreg[U/Ty − 1] := 0.
  // part one
  for J:=0 to s − 1 step Tx
    if j < i − 1 then
      areg[k] := A(j, i + k + d),
      yreg[k] += areg[k]*x(j), and
      wreg := ∑k areg[k]*x(i + k) for k ∈ [0, U/Ty).
    endif
    get_Ticket( Ticket )
    wreg := sumup wreg through Ty.
    if j < i − 1 then
      v(j) += wreg.
    endif
    release_Ticket( Ticket ), and Ticket++.
  endfor
  // part two
  for J:= thID to U do
    shm[thID][J] := shm[J][thID] := A(i + thID, i + J).
  endfor
  synchthreads
  if thID < U then
    yreg[k] := shm[thID][k]*x(i+k) for k ∈ [0, U/Ty).
  endif
  shm[k][thID] := sumup yreg[k]
                  through Tx for k ∈ [0, U/Ty).
  if thID < U then
    y(i+thID) += alpha*shm[thID][thID].
  endif
endkernel

kernel symv_postprocess
  // part three
  j := get_threadID().
  if j < n then
    y(j) += alpha*v(j).
  endif
endkernel
```

図 3 2 次元ブロック拡張した Atomic アルゴリズム (行列 A が上三角収納形式の場合, CUDA の記法を一部流用した簡易記述による. また, 行列の次元 n はパネル幅 U で割り切れるとして、端数処理部分は省略している.)

```
function get_blockID
  if isMasterThread() then
    c := atomicInc( &Master_blkID ).
  endif
  broadcast c of MasterThread.
  return MAX_blkID − c.
endfunction

function get_threadID
  return threadIdx.x+blockIdx.x*blockDim.x.
endfunction

function get_localID
  return threadIdx.x+threadIdx.y*blockDim.x.
endfunction

procedure get_Ticket( int *Ticket )
  if isMasterTthread() then
    while (TRUE)
      c := atomicCAS( Ticket, blkID, −1 ).
      if c = blkID break
    endwhile
  endif
  syncthreads
endprocedure

procedure relase_Ticket( int *Ticket )
  syncthreads
  if isMasterThread() then
    atomicExch( Ticket, blkID − 1 ).
  endif
endprocedure
```

図 4 Atomic アルゴリズム補助手続き

る. 先に示したように, 配列 v(i) の加算をマルチスレッドで並列処理するために part one に排他制御が必要となる. 図 3 では, mutex をエミュレートした Ticket メカニズム

- get_Ticket()
- release_Ticket()

の採用により, クリティカルセクションを制御する仕組みを実現している (図 4 に詳細疑似コードを掲載).

get_Ticket() は atomicCAS の第二引数によって変数 Ticket の値を変更できるブロックが 1 つに制限されている. 処理を完了したブロックが release_Ticket() を実行する際に次ブロックに許可を与える. そのため, クリティカルセクションを通過するブロックに決定的順序を与えることができる.

CUDA 6.0[12] の CUBLAS[1] ではアトミック演算を使用した実装が導入されている. しかしながら, 計算アルゴリズム中の非決定的振舞いの影響で, 演算順序が非決定的になり丸め誤差の範囲で実行結果が実行毎に異なるとの消極的なコメントがある. 一方, 本研究におけるアトミック演算はアクセス順序の制約を設けており, ブロック ID の値が非決定的であっても, それ以降は決定的であるので計算結果が実行毎に異ならない.

```
// void symv <T>
//       ( char, int, T, T*, int, T*, int, T, T*, int )
//
void ASPEN_dsymv ( char uplo, int n,
      double alpha, double *a, int lda,
                    double *x, int incx,
      double beta,  double *y, int incy );
void ASPEN_ssymv ( char uplo, int n,
      float alpha, float *a, int lda,
                   float *x, int incx,
      float beta,  float *y, int incy );
void ASPEN_chemv ( char uplo, int n,
      cuFloatComplex alpha, cuFloatComplex *a, int lda,
                            cuFloatComplex *x, int incx,
      cuFloatComplex beta,  cuFloatComplex *y, int incy);
void ASPEN_zhemv ( char uplo, int n,
      cuDoubleComplex alpha, cuDoubleComplex *a, int lda,
                             cuDoubleComplex *x, int incx,
      cuDoubleComplex beta,  cuDoubleComplex *y, int incy);
void ASPEN_wsymv ( char uplo, int n,
      cuddreal alpha, cuddreal *a, int lda,
                      cuddreal *x, int incx,
      cuddreal beta,  cuddreal *y, int incy);
```

図 5 本研究で開発した x-SYMV カーネルの API

2.1.3 その他

part two, three の部分は, 図 2 における対角ブロックの処理と, ベクトル v と y の和を求める最終処理にあたる. 関数 preprocess はベクトル v, y の初期化と, アトミック演算による排他処理のための初期化を行う.

2.2 複合型への対応 (template/cuComplex/dd_real)

本研究ではテンプレートの機能により, 複数型の SYMV の実装を 1 ソースファイルで管理する方式を採用している. 本方式では, ビルトイン型の double と float の区別ないコードを作成し, 型 `T` が double か float かの判別をするマクロもしくは関数定義を追加すれば十分である.

以下, 複合型である複素数と多倍長数を実装する際の技術内容をまとめる (図 5 は今回開発した x-SYMV カーネルの API をまとめたものである).

2.2.1 複素数 ([CZ]HEMV)

CUDA で複素数型を扱う場合には, 通常 cuComplex.h を呼び込み, cuFloatComplex もしくは cuDoubleComplex で typedef された float2 や double2 を用いてデータ型を扱う. しかしながら, cuComplex の実装ではソースコード上の加減乗算は関数呼び出しで対応する必要があり, 複素数化を容易にするため関数呼び出しではなく演算子のオペレータオーバーロードにより対応している. 基本的に, 共役複素数や実部虚部の扱い, 定数型変換を除けば実数版のソースコードと共通化できる.

2.2.2 4 倍精度 (WSYMV)

GPU の計算能力が増大化すると単純な演算に留まらない高密度演算に計算資源を回すことができるようになる. 本研究では, 倍精度浮動小数フォーマットでは不足する有効桁数をカバーする高精度計算フレームワークの拡張に余剰演算能力を使用する. 特に, Bailey らが提唱する DD(double double) 型 [14], [15] は高精度化への要求を倍精度演算のみで簡易に実現することのできる技術である. 既存研究として MPACK[16] の DGEMM 実装や QPBLAS-GPU[17] などが存在し, GPU に最も適した技術といえる.

DD の演算は倍精度加減乗算で構成されるが, 今回使用した Bailey のアルゴリズムでは 1DD 積和演算あたり 21 回の浮動小数点演算が必要である. 積和演算の要求 Byte/flop は 3 オペランドの LD と 1ST を含む故に (3+1)*(8*2)/21=3.0Byte/flop と評価される. この範囲では DD 演算は演算律速ではなくメモリバンド幅律速である. すなわち, 演算量の増加はメモリアクセスの裏に隠すことができ, プログラム上の DD 演算数で見た演算性能 DDFLOPS は, DD が double の 2 倍のサイズになるのと同様のメモリアクセス増加に伴う計算性能から見積もることができる. すなわち, メモリ律速の条件下では DD の理論性能上限は倍精度計算の 1/2 である.

疑似 4 倍精度 DD(double double) 型を扱う場合, 複素数型の様に typedef された double2 を用いてデータ型を扱う場合と, DD クラスを定義して内部で double2 を管理する 2 つの方法が有効である. 本研究では, 開発上の問題点により後者のクラスを使用する実装を見送り, 前者の typedef による実装 `cuddreal` を使用している (ホスト側は qd ライブラリ [14] の `dd_real` を使用する). [*2]

3. 性能自動チューニング (AT)

3.1 本実装におけるパラメタ群

本実装ではブロック形状を決定する 3 パラメタ (Tx, Ty, U) に加えて, SM(X) 上でアクティブな有効ブロック数 (以下, 「多重度」と呼ぶ) m, 更に行列データのアクセス順序を変えるパラメタ M を保持する. パラメタ m, M については先行研究 [13] に詳細が説明されている. 各パラメタの取りうる範囲や, それ以外の GPU やカーネルの種類で一意に決まるパラメタ群を表 1 にまとめた.

ブロック形状に制約条件があるため, 単純な積では算出できないが, 5 組のパラメタがとりうる数は倍精度版のもので少なくとも 3880 通りある [*3].

3.2 パラメタ探索アルゴリズム

先行研究 [13] では, パラメタ探索に 2 段階のチャンピオン方式の篩い分けと d-spline[18] によるデータ補間によっ

[*2] なお, typedef による実装は同じ型に結び付けられた cuComplex との併用ができないなどの問題点が生じるため本実装は応急処置的なものと位置付けている. 今後, 早急に DD クラスを使用した実装に切り替える予定である.

[*3] 実際はコンパイラの結果によって定まる「可能な m の組み合わせ」を考慮することになるため, 数倍にのぼることになる

表 1 チューニングに用いるパラメタ一覧

パラメタ I (探索対象)

T_x	スレッドサイズ x	$\{32, 64, ..., T_{x\max}\}$
T_y	スレッドサイズ y	$\{1, 2, ..., 8\}$
U	ブロックあたりパネル幅	$U/T_y \in \{3, 4, ..., 32\}$
M	ストリームオーダー	$\{1, 2, ..., 10\}$

制約条件

i) 96 ($3 \times$ WarpSize) $\leq T_x * T_y \leq T_{x\max}$,
$T_{x\max} := \{288$ (D, W, Z, C), 320 (S)$\}$.
ii) $U \leq T_x$.

パラメタ II (GPU アーキテクチャで固定 boolean 値)

USE_VOLATILE	volatile の使用
USE_TEXTURE	texture memory の使用
USE_RESTRICT	const TYPE __restrict__ * による read only cache の使用
USE_LDG	ldg() による read only cache の使用

	Fermi	Kepler	Maxwell
USE_VOLATILE	1	1	1
USE_TEXTURE	1	0	0
USE_RESTRICT	0	1	0
USE_LDG	0	0	1

てデータサンプリングの回数を削減していた. 本研究も基本的には同じ枠組みに則りサンプリング回数を減らすが, さらに経験的に得られた条件として,

- Tx, Ty, U が同一の時, より大きな m 値を選択する.
- Tx, Ty, m が同一の時, より大きな U 値を選択する.

上 2 条件を加えて, パラメタの探索範囲を狭める.

表 2 が各 GPU アーキテクチャにおいて得られた SSYMV の Top 5 パラメタである. 実際は, 第一段階のパラメタ篩い分けで上位 20 カーネルを取得している. 続いて, d-spline を用いてこの Top20 のカーネルのより高詳細なデータ補間を行い, 各次元で最高性能を示すパラメタを決定する. これは SYMV カーネル構築時に決定されるため, 表 3 のような if-then ルールとして作成される. カーネル実行時にはこの if-then ルールから適切なパラメタが選択される.

表 2 から, Ty が 2 以上のものが上位に入っていることが判る. これは, 2 次元スレッドブロックがよりよい性能を示すことを意味しており, 今回の新実装が著者らの従来実装よりも高い性能を示すものと期待される. また, GPU アーキテクチャによって上位パラメタの傾向が異なっている. 詳細な分析が必要であるが, GPU の世代が同じものには良好パラメタの組に共通性・相関性が見られるようである. この分析結果も, 新しいアーキテクチャの出現時にパラメタ空間探索の削減に利用できる可能性がある.

4. 性能測定

前節で開発したチューニング済みの SYMV カーネルの性能測定を表 4 にある 4GPU のもとで実施した. 序章から説明しているように, SYMV の性能はメモリバンド幅に律速されるため, それぞれの要求 Byte/flop 値とメモリバンド幅 B/s から理論的な性能上限値を求めることができる. SYMV のそれぞれの Byte/flops 値は W 4(=(8*2)/4), D 2(=8/4), S 1(=4/4), Z 1(=(8*2)/(4*4)), C 0.5(=(4*2)/(4*4)) であるので, この値でメモリバンド幅を割ればよいことになる. 例えば Titan Black を例にすれば

- W(cuddreal: 疑似 4 倍精度実数): 84GFLOPS
- D(double: 倍精度実数): 168GFLOPS
- S(float: 単精度実数): 336GFLOPS
- Z(cuDoubleComplex: 倍精度複素数): 336GLOPS
- C(cuFloatComplex: 単精度複素数): 672GFLOPS

のように定まる. 性能比が W:D:S:Z:C=1:2:4:4:8 に近くなることが理想的な状態といえる.

4.1 [DS]SYMV の性能

図 6 から図 9 に, 表 4 で示した GPU で自動チューニングした [DS]SYMV の性能 (GFLOPS) をプロットした. 比較のため, BLAS の CUDA 実装で著名な CUBLAS 6.5[1], KBLAS 1.0[10], MAGMABLAS 1.5.0(beta3)[2] も同じ条件で測定した. 本研究で開発した SYMV カーネルは, 倍精度版では 4GPU ともに最速であり, 10 から 25% 程度高速である. 一方, 単精度では十分速いが, Titan Black における CUBALS6.5 との性能差は殆どない状況である. NVIDIA 社は最新の GPU である Titan Black に特化したチューニングを施している可能性がある.

各 GPU のメモリバンド幅換算の性能をまとめると以下の様になる.

- GTX580: D(148GB/s=77%), S(149GB/s=78%)
- K20c: D(134GB/s=64%), S(131GB/s=63%)
- Titan Black: D(220GB/s=65%), S(205GB/s=61%)[*4]
- GTX750Ti: D(68GB/s=78%), S(74GB/s=85%)

メモリバンド幅に対して少なくとも 61%を達成しており [*5], 他の実装よりも高速である. また, 他の実装ではブロックサイズで行列サイズが割り切れるときに特別なカーネルを使い分けている. そのため, 性能測定点が 2 曲線上に分布する形になる. 本実装ではその様な使い分けを行わないため, 1 曲線のみ表れている. また, 性能の揺れは認められるものの全般的に安定である.

4.2 WSYMV, [CZ]HEMV の性能

図 10 に GeForce GTX Titan Black における, 疑似 4 倍精度版 WSYMV, 単精度複素数版 CHEMV, 倍精度複素数版 ZHEMV の性能をプロットした. 測定結果は WSYMV を除いて要求 Byte/flop から説明できる結果であり, 性能

[*4] 6000MHz 動作をピークと考えれば, それぞれ 76%, 71%である.
[*5] bandwidthTest による実測では, GTX580: 170GB/s, K20c: 146GB/s, Titan Black: 229GB/s, GTX750Ti: 67.3GB/s である.

表 2 SSYMV における各 GPU アーキテクチャの Top 5 パラメタ (Tx, Ty, U, m, M)

kernel ID	GTX580	K20c	Titan Black	GTX750Ti
1.	(64, 2, 42, 4, 0)	(64, 4, 48, 4, 0)	(64, 4, 48, 4, 0)	(96, 1, 23, 8, 0)
2.	(64, 2, 42, 4, 1)	(96, 3, 27, 4, 1)	(64, 5, 60, 3, 0)	(64, 2, 28, 8, 0)
3.	(32, 8, 32, 2, 9)	(96, 3, 51, 3, 0)	(64, 4, 44, 4, 1)	(32, 8, 32, 2, 9)
4.	(64, 4, 64, 2, 1)	(64, 2, 34, 7, 0)	(96, 3, 27, 4, 0)	(64, 2, 28, 8, 3)
5.	(96, 3, 36, 2, 0)	(64, 4, 44, 4, 1)	(128, 2, 36, 3, 3)	(64, 2, 36, 7, 1)

表 3 各 GPU でのカーネル選択ルール (ID=0 は L+U アルゴリズムを表す. if-then ルールが長いものは途中を省略した)

GTX580	K20c	Titan Black	GTX750Ti
if ($1 \leq n < 1600$) {	if ($1 \leq n < 1771$) {	if ($1 \leq n < 2098$) {	if (($1 \leq n < 470$) {
ID=0;	ID=0;	ID=0;	ID=0;
} elsif ($1600 \leq n < 1842$) {	} elsif ($1777 \leq n < 2989$) {	} elsif ($2098 \leq n < 2172$) {	} elsif ($470 \leq n < 2610$) {
ID=16;	ID=6;	ID=6;	ID=9;
} elsif ($2166 \leq n < 1842$) {	} elsif ($2989 \leq n < 3565$) {	} elsif ($2172 \leq n < 4378$) {	} elsif ($2610 \leq n < 2904$) {
ID=13:	ID=5;	ID=14;	ID=1;
...	} elsif ($3565 \leq n$) {	...	...
} elsif ($5790 \leq n$) {	ID=1;	} elsif ($32412 \leq n$) {	} elsif ($19550 \leq n$) {
ID=1;	}	ID=1;	ID=6;
}		}	}

表 4 実験に使用した GPU の諸性能ならびにホスト CPU のハードウェア/ソフトウェア環境 (*3 カタログスペック上は 7000MHz であるが, GPUBoost の影響で負荷時には 6000MHz で動作する. 6000MHz 時のメモリバンド幅は 288GB/s.)

	GTX580	Tesla K20c	Titan Black	GTX750Ti
Compute Capability	2.0	3.5	3.5	5.0
GPU Clock (MHz)	1544(boost NA)	706(boost NA)	889(boost 980)	1020(boost 1085)
Multiprocessors	16	13	15	5
CUDA Cores	512	2496	2880	640
Memory Capacity (MB)	1536	5120	6144	2048
Memory Clock (MHz)	4008(384bit)	5200(320bit)	7000(384bit)*6	5400(128bit)
Memory Bandwidth (GB/s)	192	208	336	86.4
ECC Support	NA	Enabled	NA	NA
Host	(a)	(b)	(c)	(a)

	Host (a)	Host (b)	Host (b)
CPU	AMD FX-8120	Intel Core i7-3930K	Intel Core i7-3930K
CPU Core 数	8	6	6
CPU Clock (GHz)	3.1	3.2	3.2
Memory Capacity (GB)	16	16	16
Linux Kernel version	3.6.11-4	3.11.10-100	3.11.10-100
CUDA Version	6.5	6.5	6.5
Driver Version	340.29	340.29	340.29
GNU gcc Version	4.6.3	4.7.2	4.7.2

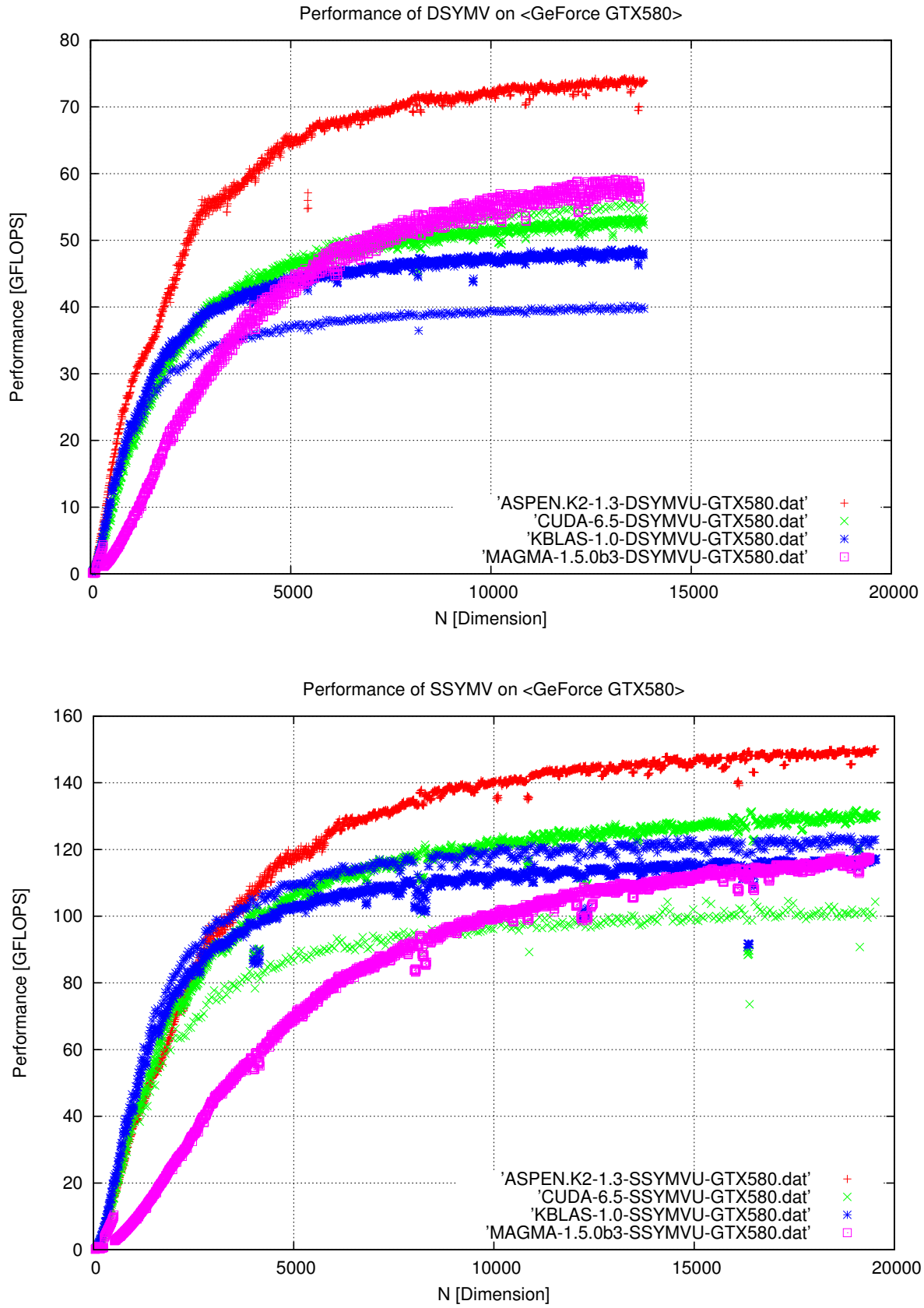

図 6　GeForce GTX580 での SYMV の性能 (上: DSYMV 倍精度, 下: SSYMV 単精度, それぞれ行列は 8 次元毎に測定)

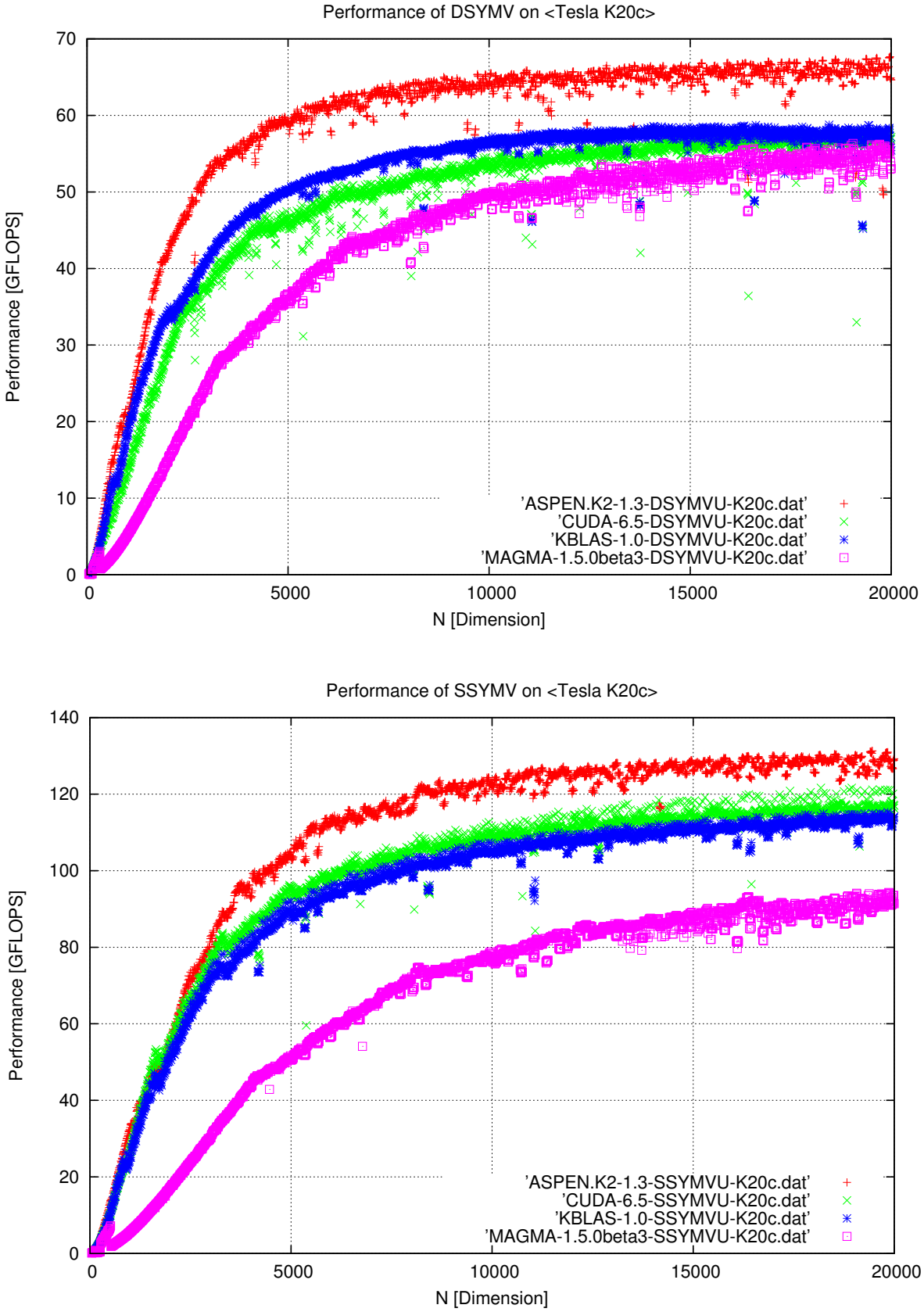

図 7　Tesla K20c での SYMV の性能 (上: DSYMV 倍精度, 下: SSYMV 単精度, それぞれ行列は 8 次元毎に測定)

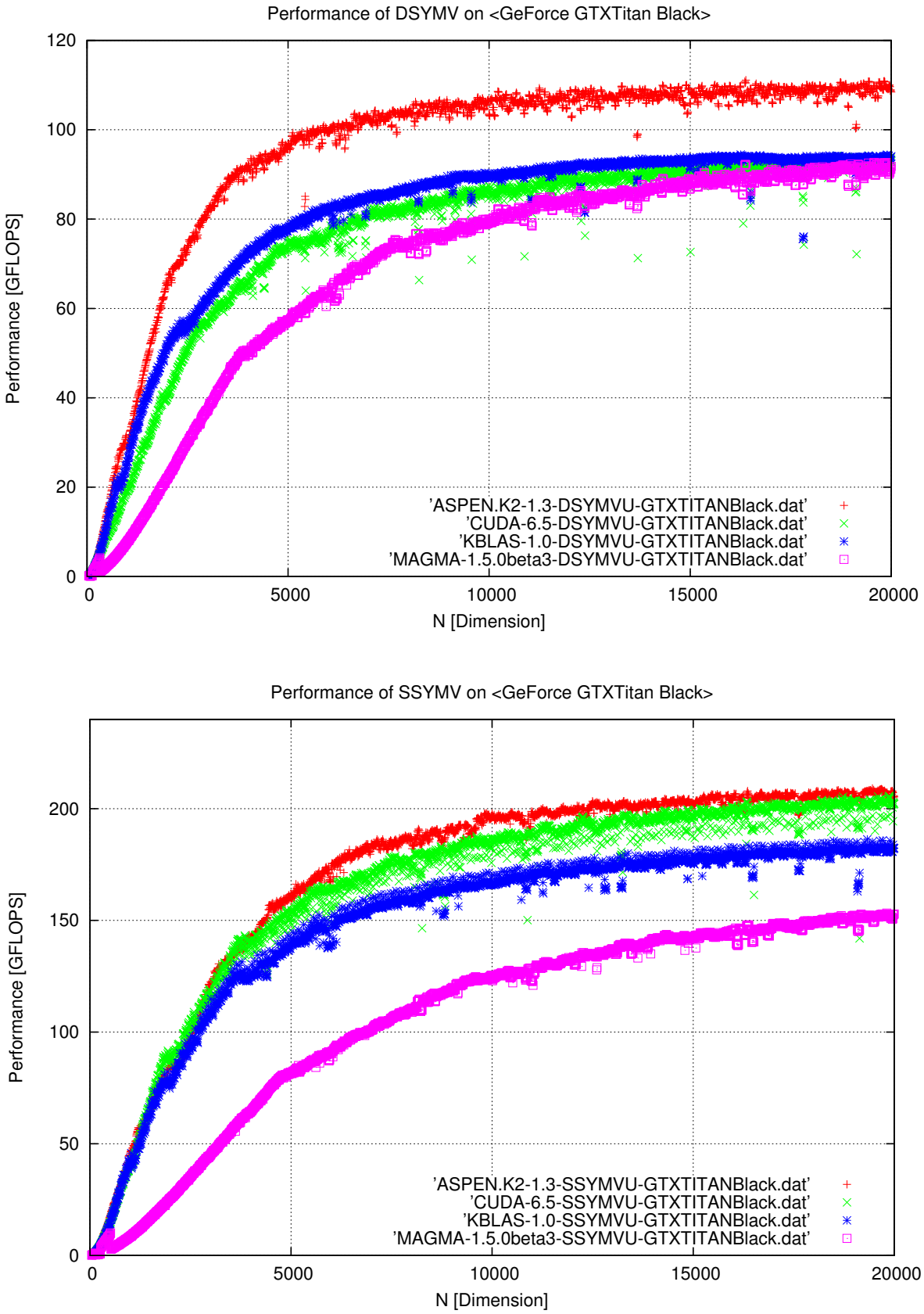

図 8　GeForce GTX Titan Black での SYMV の性能 (上: DSYMV 倍精度, 下: SSYMV 単精度, それぞれ行列は 8 次元毎に測定)

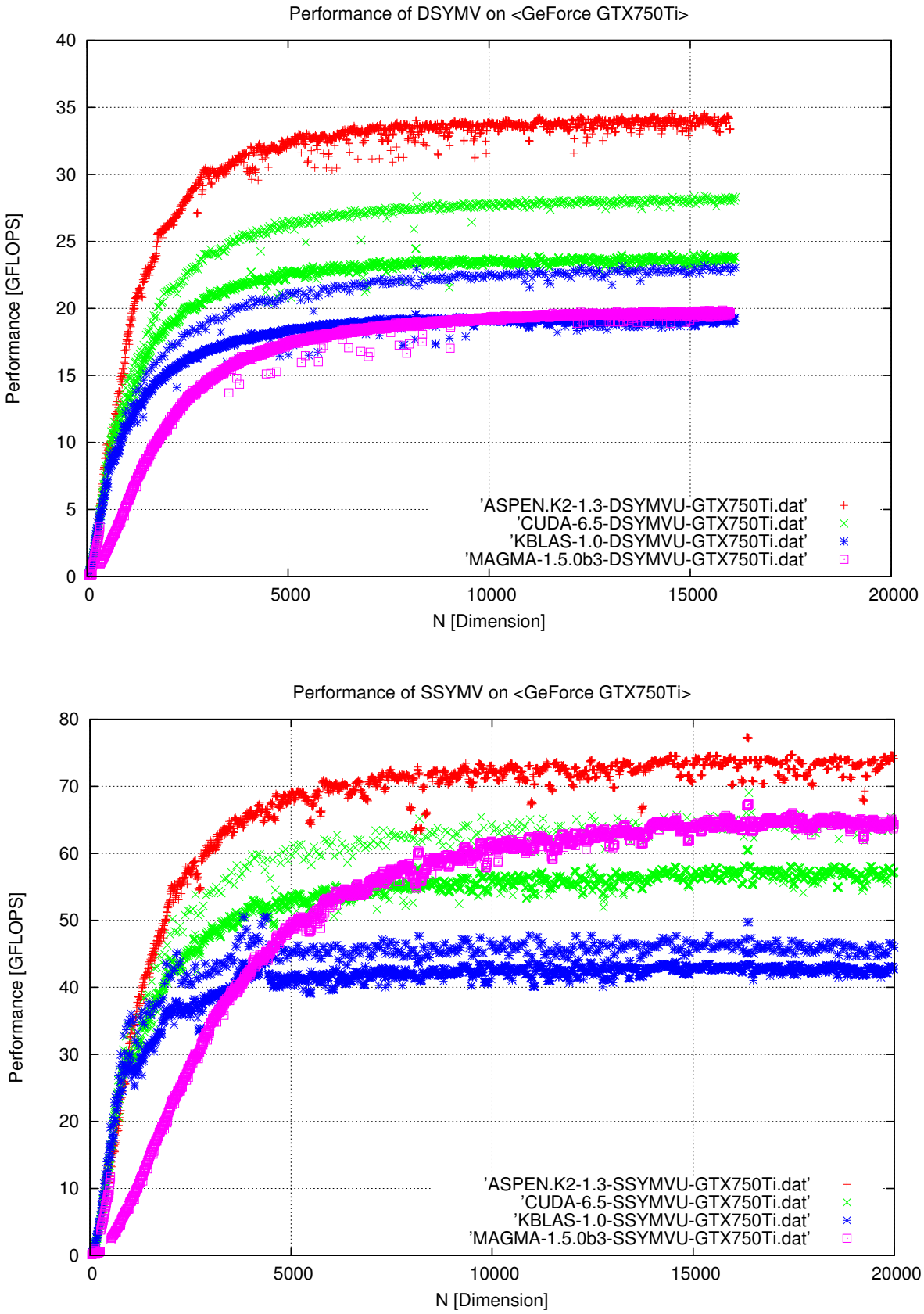

図 9　GeForce GTX750Ti での SYMV の性能 (上: DSYMV 倍精度, 下: SSYMV 単精度, それぞれ行列は 8 次元毎に測定)

が D:S:Z:C=1:2:2:4 となっている. つまり, メモリバンド幅を [DS]SYMV 同様に十分に生かせていると結論付けられる. 一方, WSYMV は. DSYMV の 1/2 の 50GFLOPS が期待されるところであるが, その 40% にも達しておらず, 性能面で若干劣ることが読み取れる. この性能低下の主たる理由は

(1) 本実装のチューニングはまだ実験段階であり, [SD]SYMV ほど広いパラメタ範囲を探索していない,
(2) 探索候補を決定する際にレジスタスピルによる影響を加味していない,
(3) nvcc コンパイラの生成する DD クラスオブジェクトの最適化が十分でない,

などが考えられる. チューニングの余地が十分にあり, 今後の更なる高速化が望める.

5. まとめ

本研究では, SYMV の対称性を利用する際に, アトミック演算を用いた mutex の実装を工夫することによりアクセス順制御を実現し「実行毎に丸め誤差の範囲で演算結果が異なる」という現象を回避した. また , 既存研究ではスレッドブロック形状を 2 次元に拡張し, 使用コア数を増加させる改良を施した. 自動チューニング技術による最適パラメタを組み込み高性能カーネルの構築に成功している. 世代の異なる 4GPU 環境での実測からも, 他の CUDA BLAS 実装と比較して高い性能を示している. さらに, 実数 (単精度や倍精度) 以外の数値フォーマットである複素数 (単精度 , 倍精度) ならびに疑似四倍精度 DD(double-double) フォーマットに対しても , 同様のアプローチによりカーネル実装を行った. 本研究で培われた実装技術は Level 2 BLAS の他の関数にも適用が可能であり, 高性能・高精度 GPUBLAS の開発に展開していくことが今後の課題といえる.

最後に, 本研究は科研費新学術領域研究 (課題番号: 22104003) の支援を受けている. また, 本研究で開発したカーネル関数のうち [DS]SYMV は自動チューニング機能を有した高性能 Level-2 CUDA BLAS カーネル ASPEN.K2 に収録されており, (http://www.aics.riken.jp/labs/lpnctrt/ASPENK2.html より公開中). WSYMV ならびに [CZ]HEMV は性能を更にチューニングし公開予定である.

参考文献

[1] NVIDIA Corporation, The NVIDIA CUDA Basic Linear Algebra Subroutines, http://developer.nvidia.com/cublas
[2] Innovative Computing Laboratory, University of Tennessee, Matrix Algebra on GPU and Multicore Architectures, http://icl.cs.utk.edu/magma
[3] Sørensen, H. H. B., Auto-tuning Dense Vector and Matrix-Vector Operations for Fermi GPUs, Parallel Processing and Applied Mathematics, LNCS 7203 (2012) 619–629.
[4] Sørensen, H. H. B.. Auto-Tuning of Level 1 and Level 2 BLAS for GPUs, Concurrency Computat.: Pract. Exper., Wiley (2012) 1183–1198.
[5] GPUlab: GLAS library version 0.0.2, http://gpulab.imm.dtu.dk/docs/glas_v0.0.2_C2050_cuda_4.0_linux.tar.gz
[6] 今村俊幸, CUDA 環境下での DGEMV 関数の性能安定化・自動チューニングに関する考察, 情報処理学会論文誌コンピューティングシステム, Vol.4, No.4 (Oct. 2011) 158–168.
[7] Abdelfattah, A., Keyes, D., and Ltaief, H., KBLAS: High Performance Level-2 BLAS on Multi-GPU Systems, http://ondemand.gputechconf.com/gtc/2014/poster/pdf/P4168_KBLAS_GPU_ computing_optimization.pdf, GTC2014 (2014).
[8] Imamura, T., ASPEN-K2: Automatic-tuning and Stabilization for the Performance of CUDA BLAS Level 2 Kernels, 15th SIAM Conference on Parallel Processing for Scientific Computing (PP2012), http://www.siam.org/meetings/pp12/
[9] Nath, R., Tomov, S., Dong, T. T., and Dongarra, J., Optimizing Symmetric Dense Matrix-vector Multiplication on GPUs, in Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC'11 (2011) 6:1–6:10.
[10] Abdelfattah, A., Keyes, D., and Ltaief, H., KAUST BLAS (KBLAS), http://cec.kaust.edu.sa/Pages/kblas.aspx
[11] Imamura, T., Yamada, S., and Machida, M., A High Performance SYMV Kernel on a Fermi-core GPU, High Performance Computing for Computational Science — VECPAR 2012, LNCS 7851 (2013) 59–7.
[12] NVIDIA Corporation, CUDA C Programming guide, http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf (2014).
[13] 今村俊幸, 内海貴弘, 林熙龍, 山田進, 町田昌彦, Fermi, Kepler 複数世代 GPU に対する SYMV カーネルの性能チューニング, 情報処理学会研究報告,「ハイパフォーマンスコンピューティング（HPC)」, Vol. 2012-HPC-138, No. 8 (2012) 1–7.
[14] Hida, H., Li, X. S., and Bailey, D. H., Quad-double arithmetic: Algorithms, implementation, and application (Oct 2000), Online PDF http://www.davidhbailey.com/dhbpapers/quad-double.pdf
[15] Bailey, D. H., and Borwein, J. M., High-Precision Computation and Mathematical Physics, texttthttp://crd.lbl.gov/~dhbailey/dhbpapers/dhb-jmb-acat08.pdf
[16] Nakata, M., The MPACK (MBLAS/MLAPACK); a multiple precision arithmetic version of BLAS and LAPACK, http://mplapack.sourceforge.net/
[17] 佐々成正, 山田進, 町田昌彦, 今村俊幸, 奥田洋司, QPBLAS-GPU の開発と性能評価第 18 回計算工学会講演会計算工学会論文集, Vol. 18, D-13-5 (2013).
[18] 田中輝雄, 数値計算ライブラリを対象としたソフトウェア自動チューニングにおける性能パラメタ推定法に関する研究, 電気通信大学博士学位論文 (2008).

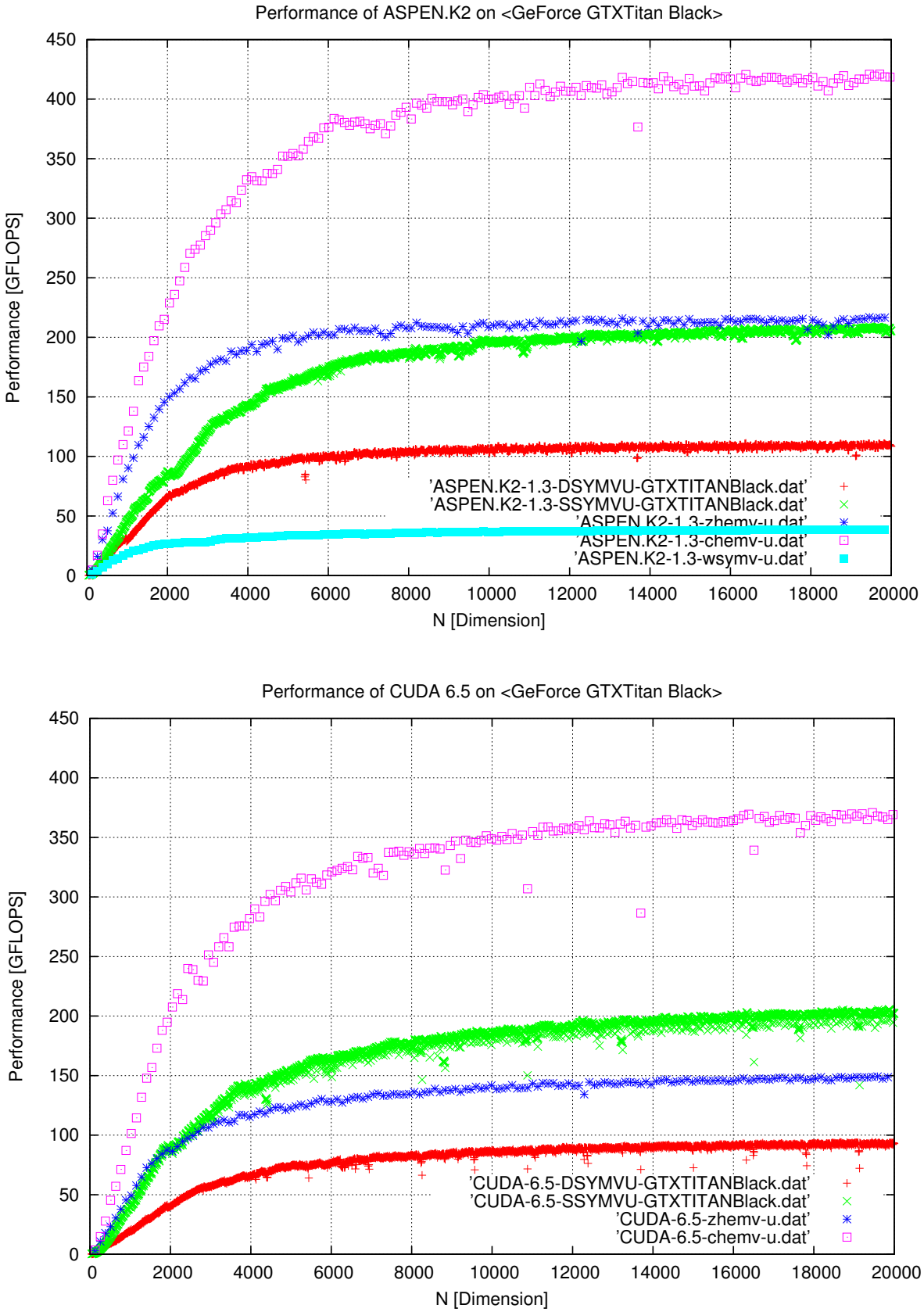

図 10 x-{SY|HE}MV の性能 (GeForce GTX Titan Black, 上: ASPEN.K2, 下: CUDA6.5, [DS]-SYMV は 8 次元毎, WSYMV,[CZ]-HEMV は 32 次元毎にて測定, WSYMV の性能は DD 演算で見た値であり所謂 DDFLOPS で測ったものである)