Vol.2015-HPC-151 No.13
2015/9/30

SYMV・GEMVルーチン群のマルチGPU化とその評価

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：本研究では, 我々がこれまでに CUDA single GPU 向けに開発してきた ASPEN.K2(SYMV) ならびに MUBLAS(GEMV) をマルチ GPU に拡張し, パラメーター自動チューニングの技法により性能最適化を行う, SYMV ならびに GEMV を Tesla K20Xm 最大4 GPU で動作する拡張を施し、現時点での試験実装方式の整理と性能予備評価を実施する. マルチ GPU 化の問題点や性能向上技術, 将来の技術動向を勘案したマルチ実装向け BLAS, 特に Level-2 BLAS カーネル開発の知見集積を目的とする.

1. はじめに

我々はこれまでに, CUDA 環境 [1] での高度な最適化手法に加えて自動チューニング技術の適用により CUBLAS[2] や MAGMABLAS[3] を凌ぐ CUDA 環境向けの Level-2 BLAS ソフトウェア ASPEN.K2&MUBLAS の開発を進めている ([4], [5] など). 本研究で対象とする GEMV や SYMV はベンダ提供の CUBLAS での性能が十分安定してよいものではないことが多い. これらは, メモリインテンシブ型のカーネルであることが知られており, ハードウェア上に実装されるメモリバンド幅に性能が制限される. GPU の場合は, グローバルメモリの GDDR5 等の転送性能上限で性能が縛られる. 一般に GPU のグローバルメモリの総帯域はホスト CPU 上のメモリよりも広く, GPU メモリ上で演算が完結する場合 (ホスト–デバイス間の転送が無視できる場合), CPU よりも数倍の高性能が期待できる. これまでにも, HPC 研究会 ([6], [7], [8], [9] など) や他の国際会議で研究経過報告を実施している. CPU ホスト上の狭メモリ帯域の問題を**ブロック化**, つまり Level-3 BLAS への書き換えによって対応してきた背景はあるものの, 近年の 3 次元積層メモリなどの広帯域メモリの登場で, Level-2 BLAS の実装方式に関する知見の積み上げの重要性はより高まってきている.

現在の GPGPU の技術課題を見ると, シングルノード上での GPU デバイス数の増加が挙げられる. 所謂マルチ GPU である. NVIDIA と IBM は NVLINK 等の活用により, より広帯域な GPU デバイス間技術を近い将来実現し, シングルノード上の高集積 GPU サーバーを開発することを表明している. ハードウェア周りの高性能広帯域デバイスとしての整備は着々と進んでいると考えられる.

一方, ソフトウェア的な側面, 特に著者らが最も興味を持つ**数学ソフトウェア**の面からは, 上述の面に対応する技術は提案されつつあるが, 必ずしも性能面や利用者の利便性の観点からは十分なものにまだ達していない可能性がある. 数学ソフトウェアのプリミティブの一つである数値線形計算では BLAS の性能と開発利便性が重要視される. 我々のこれまでの CUDA 環境向けの BLAS 開発はシングル GPU・スタンドアロン環境に閉じていた. 高性能な BLAS は NVIDIA 社, MAGMABLAS(テネシー大学), KBLAS(KAUST) などで実施されてきており, シングル GPU から始まり, 必ずしもすべてのカーネル関数が対象ではないが, マルチ GPU での機能実現がなされている. これらの機能実現のためのプログラミング環境としては CUDA が提供するストリーム機能の並列動作によるところが多いが, 近年 GPU Direct のノード内 GPU 間通信やノード間 RDMA 通信なども CUDA 世代の進化に応じて機能追加がなされている. 更に, Unified Memory management も CUDA6 以降サポートが逐次始まり, 主にホスト-デバイス間通信などの隠蔽がハードウェア的にも実現可能になるとの方向にある.

本報告の目的は, 1) マルチ GPU 上での BLAS カーネル構築の問題点の蓄積, 2) シングル GPU 上で培った自動チューニング技術の使用可否の検討などについて予備検討を実施するとともに, その性能測定結果の速報を行うことにある.

[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

2. マルチ GPU 環境

本研究では同一ノード上でアクセス可能な複数 GPU を**マルチ GPU** と呼ぶ. 一般に, 同一マザーボード上の PCI Express 等のバスに挿された複数の GPU がこれにあたる. 完全に同型 GPU で構成される場合には「ホモジニアス」, 僅かでも異なる GPU が含まれる場合は「ヘテロジニアス」環境にあるとする. 以下, 本稿ではマルチ GPU 環境における GPU 数を ngpus とする.

2.1 CUDA でのマルチ GPU

CUDA 環境における基本的なマルチ GPU カーネルの動作手順は以下のようになる.

(1) 各 GPU コンテクストに対応するストリームハンドラ stream の生成.

```
cudaStream_t stream[MAX_NGPUS];
for(i=0;i<ngpus;i++){
  cudaSetDevice(i)
  cudaStreamCreate(&stream[i]);
}
```

(2) 各 GPU デバイス上のメモリ確保と設定

(3) GPU 毎にカーネル関数の非同期並列呼び出し. コンテキストに対応するストリームハンドラ stream をストリーム引数に渡す.

```
for(i=0;i<ngpus;i++){
  cudaSetDevice(i)
  kernel <<< thread, grid, shmem, stream[i]
>>> (args[i], ..., i, ngpus );
}
```

(4) 各コンテキストの同期

```
for(i=0;i<ngpus;i++){
  cudaSetDevice(i)
  cudaStreamSynchronize(stream[i]);
}
```

など

(5) 各 GPU デバイス上のメモリ解放 ストリームの解放

```
for(i=0;i<ngpus;i++){
  cudaSetDevice(i)
  cudaStreamDestroy(stream[i]);
}
```

上記の様に, 各 GPU デバイスのコンテキスト切り替えを行い, 対応するホスト/デバイス側の制御を行う. 非同期のカーネル関数呼び出しの同期処理を行えばよい. カーネル関数内部での特別な記述は不要であるが、上例では使用する GPU 数と GPU ID をカーネル関数に渡している. 必要に応じて GPU ID を参照して処理を切り替える形にすればよい.

本研究では, シングル GPU プログラムに対して上記なような処理の流れへの修正と, デバイスデータの GPU 間分散, GPU ID に応じた処理対象部分変更への対応を主に実施する.

2.2 マルチ GPU 対応の BLAS

2.2.1 CUBLAS

CUBLAS[2] は NVIDIA 社が開発する **CUDA SDK**[1] に標準で提供されている BLAS の CUDA 実装である. 基本的にシングル GPU 向けの実装であるが, 近年マルチ GPU 向けの実装 CUBLAS-XT が提供されている [10]. しかしながら, 現状は level-3 カーネルのみサポートされるのみである. また, 汎用インターフェイス (CUBLAS の thnking モードのマルチ GPU 版に相当) として NVBLAS も存在する. Level-3 カーネルでの効果は期待できるものの Level-2 以下のカーネルではホスト-デバイス間通信が陽に含まれてしまうためマルチ GPU 化による大きなアドバンテージはない状況である.

2.2.2 MAGMABLAS

MAGMA(テネシー大学)[3] のサブセットで開発されている MAGMABLAS は上位の MAGMA ライブラリが必要とするカーネルを中心にサポートされているが, 特に, 固有値計算に必要な SYMV, R2K, HEMM 関数がマルチ GPU 対応されている. 現在 MAGMA の最新版は 1.6.2 である.

2.2.3 KBLAS

KBLAS[11], [12](KAUST) では Level-2 BLAS を中心に実装が進んでおり, 多くの関数でマルチ GPU 対応がなされている. GEMV と SYMV は非常に高い性能を達成しており, 我々の研究でも比較対象とすることが多い. 現在, バージョン 1.3-beta が最新版である.

2.2.4 並列化・実装方式とデータ分割

CUBLAS-XT の実装方法の詳細は不明だが, ホワイトペーパーにはタイリングによるデータ分割を基本とした小規模行列行列積を中心に構成されていると述べられている.

MAGMABLAS, KBLAS はブロックサイクリック分割方式で実装されている. GEMV は縦長行列に分割し, 個々の GPU デバイス上で対応する行列データとベクトルとの積を計算するカーネルを呼び出す形で構成されている. SYMV も基本的には同様でタイル単位の行列ベクトル積を組み合わせて構成されているが, 行列フォーマットが上三角, 下三角であるために対角以外のブロック行列で折り返し (右からのベクトル積と左からのベクトル積の同時計算に相当) を実現している. MAGMABLAS と KBLAS の主な違いは, 前述の左右ベクトル積の総和計算の方式である. MAGMABLAS は各スレッドブロック間の総和を作業配列を介して実行する [13] のに対して, KBLAS はアトミック演算で実装されている [11], [12]. MAGMABLAS の方式は

総和用のカーネルを立てる必要があり, メモリ使用量と計算時間の面で KBLAS に対して不利である. 後述の性能比較ではより高性能な KBLAS を比較対象とする.

3. SYMV(ASPEN.K2)& GEMV(MUBLAS) のマルチ GPU 化

3.1 CUDA-BLAS 実装の方針

本研究では GEMV と SYMV カーネルのマルチ GPU 化実装を施すが, GEMV は MUBLAS の, SYMV は ASPEN.K2 のカーネルコードをベースとする. また, データ分散方式は CUBLAS-XT など他のマルチ GPU 版 BLAS 実装に合わせるべきであるが, 本研究では MUBLAS, ASPEN.K2 共に自動チューニングの性質を持ち合わせることと, マルチ GPU 実装上の問題点をある程度洗い出すことも考慮して, データ分散はカーネル関数毎に適切なものをまず選択して実装を進める.

3.2 SYMV(ASPEN.K2)

ASPEN.K2 の SYMV のマルチ GPU 化について説明する. ASPEN.K2 で実装されている SYMV は, ブロック幅 U でスライスした列ブロックを単位として, 「対角ブロック」, 「副対角ブロック」, 「左右ベクトル積の総和」の三つの処理を実施する. U は性能チューニングのパラメタであるので, 任意に選択したい. 基本的に対角ブロックに対応する部分行列の形状が容易に判定できれば実装は困難ではない. 本研究では実装が容易なサイクリック分割を採用した場合のコーディングの注意部分を考察する. 入力ベクトル x と y はカーネル関数呼び出し時点で全 GPU がコピーを保持していることとする.

3.2.1 対角ブロックのアクセス

ngpus で列をサイクリック分割したとき, 対角ブロック内のデータへのアクセスが例外処理となる. ngpus=1 の時の U × U ブロック内の上三角データへの処理は以下の様になっている.

```
for (j_=0; j_<U; j_++) { j=row+j_;
   for (i_=0;i_<U;i_++) { i=row+i_;
      A_reg[i_] = (j < i ? A(j,i) : 0);
   }
   ...
}
```

ngpus 個のマルチ GPU 環境では, 配列 A の row 列目以降の列アクセスは以下の様に ngpus 間隔になる.

```
for (j_=0; j_<=(U-1)*ngpus; j_++) { j=row+j_;
   for (i_=0; i_<U; i_++) { i=row+i_*ngpus;
      A_reg[i_] = (j < i ? A(j,i) : 0);
   }
  ...
}
```

また, 実際に行列 A のデータは一部の列しか保持されないのでローカルメモリ上では A(j,i) は Alocal(j,i/ngpus) のように修正することになる. ローカルメモリ上の上下 ngpus 倍された上下に長い ((U-1)*ngpus+1) × U ブロック内の上三角データになる (**図 1**). なお, 上記コードを CUDA 化すると以下のようになる. (Tx,Ty) はスレッド形状, Lda は行列 A の整合寸法である.

```
row_=row+threadIdx.x;
col_=row+threadIdx.y*(U/Ty);
for (j_=0; j_<=(U-1)*ngpus; j_+=Tx) {
   j=row_+j_; mask=(j<=(U-1)*ngpus);
   Alocal=A+j+(col_/ngpus)*Lda;
   if ( mask ) for (i_=0; i_<(U/Ty); i_++) {
      i=col_+i_*ngpus;
      A_reg[i_] = (j < i ? Alocal[Lda*i_] : 0);
   }
   ...
}
```

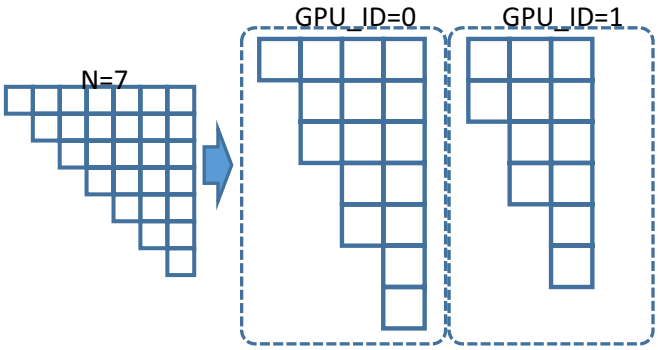

図 1 上三角行列フォーマットの 2GPU サイクリック分割例

3.2.2 副対角ブロックのアクセス

副対角ブロックへのアクセスは対角ブロックよりも容易である. 単に, 列アクセスが ngpus 間隔で, ローカルメモリ上 Alocal(j,i/ngpus) のようになる点を注意すればよい.

3.2.3 左右ベクトル積の総和

左右ベクトル積の総和計算は通常スレッドブロック間の依存関係があるため, 何らかの排他制御が必要となる. 前述のように MAGMABLAS は work 配列を導入, KBLAS はアトミック演算で実現している. ASPEN.K2 の version1.5 以降では, ブロック間同期をエミュレートする形で実現していたが [14], 今回はマルチ GPU 版の試験実装で動作が最も単純な version1.3 当時の手法を採用した. 実際は総和専用のカーネル関数を用意し 2 つのカーネル関数に分けることで, 自明な同期をそこで保証する形をとった. 複数カーネル関数の起動は性能面で不利なため, 将来的には version1.5 以降の実装方式を採用できるようにする必要があると認識している.

3.3 GEMV(MUBLAS)

MUBLAS の GEMV（$y = \alpha Ax \times \beta y$）のマルチ GPU 化について説明する．非転置モードの GEMV-N の場合，行列 A は GPU 数で行方向にブロック分割して保持し，ベクトル x と y は全 GPU がベクトル全体のコピーを持つ．各 GPU は行方向に細長い GEMV を計算することで，ベクトル y の部分ベクトルを計算する．この実装方式ではシングル GPU 向けの GEMV ルーチンをそのまま流用できるため，カーネルコードに手を加える必要はない．行列 A が m×n 行列，GPU 数が ngpus である場合に，GPU ID が 0〜ngpus-2 番の GPU は ceil(m/ngpus) 行 × n 列，ngpus-1 番の GPU は m-ceil(m/p) 行 ×n 列の計算を担当する．ngpus=4 の場合を**図 2** に示す．転置モードの GEMV-T の場合も行列の格納方向が異なるだけで，マルチ GPU 実装は GEMV-N の場合と同様である．

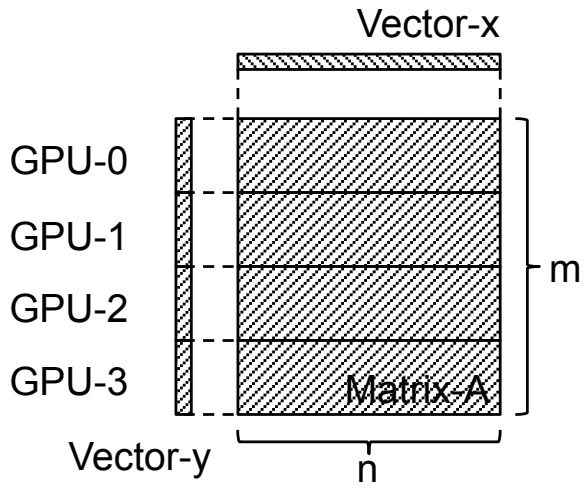

図 2 4GPU を使った GEMV の計算方式の例

4. 性能評価

4.1 マルチ GPU 性能諸元

表 1 に本実験で使用した HOKUSAI Greatwave の諸性能ならびにソフトウェア情報を示す．

4.2 SYMV

図 3 は，Tesla K20Xm を 1〜4 台使用した場合の，SSYMV および DSYMV の上・下三角モード（SYMV-U, SYMV-L）の性能を示している．なお，原稿執筆時点では ASPEN.K2 の下三角モード (SYMV-L) の実装は完了していないため，KBLAS の性能のみを参考値として示す．KBLAS の性能は KBLAS 1.3-beta のマルチ GPU 版 SYMV (`kblas_xsymv_mgpu_async`）の性能である．なお，これらの性能にはホスト-デバイス間のデータ転送時間は含まれていない．行列を列方向に分割しているため，全 GPU の計算結果を統合するためベクトル y を GPU 間もしくは CPU 上で足し合わせる必要があるが，この時間は測定に含めていない．

GPU 間でのベクトル y の総和計算を除けば，SYMV は GPU 間で通信を必要としないため，GPU 数に比例した性能向上が期待される．実測ではほぼその傾向に近い数値が得られている．カーネルにはチューニング可能なパラメタが含まれているが，本実験では 4GPU で実行する際の GPU ID=0 が担当する部分を最速にするパラメータを探索して使用している DSYMV-U では ASPEN.K2 のピーク性能は KBLAS よりも高いものの，性能の立ち上がりでは KBLAS が優勢である．オーバヘッドが ASPEN.K2 の方が大きいことになる．SSYMV-U は十分な優位性が認められないが，数%程度高速で，オーバヘッドも DSYMV-U と同様にある．オーバヘッドが大きい原因として，KBLAS は 2 カーネル関数での実装であるのに対して ASPEN.K2 の今回の実装は 3 カーネル関数での実装になっている点が挙げられる．カーネル関数呼び出しコストが影響しているものと考えられる．

また，ASPEN.K2 は 3GPU 以上でランダムな性能の揺れが確認できる．特に，単精度 SSYMV-U は変動幅が大きい，個々のカーネルは 1GPU とほぼ同等の動作をしているはずであり，OS ジッターの影響を受けていることになると予想される．しかしながら，KBLAS ではその様な性能の揺れが殆ど見られないため今後詳細を調査する必要がある．

4.3 GEMV

図 4 は，Tesla K20Xm を 1〜4 台使用した場合の，SGEMV および DGEMV の非転置・転置モード（GEMV-N, GEMV-T）の性能を示している．行列は $N \times N$ の正方行列である．KBLAS の性能は KBLAS 1.3-beta のマルチ GPU 版 GEMV（`kblas_xgemv_mgpu_async`）の性能である．SYMV 同様，ホスト-デバイス間のデータ転送時間は含まれない．また，全 GPU の計算結果を統合するための時間も測定に含めていない．GEMV は GPU 間の通信が発生しないため，MUBLAS，KBLAS ともに GPU 台数にほぼ比例した性能向上が得られている．MUBLAS は KBLAS よりも高速かつ行列サイズに対する性能変動が少ない安定した性能を実現している．

KBLAS のマルチ GPU 実装は行列を列方向にブロックサイクリック分割しており，GPU ID によって GEMV-N ではベクトル x，GEMV-T ではベクトル y の参照アドレスが異なるため，マルチ GPU 向けにシングル GPU 向けとは異なるカーネルを用意している．KBLAS の性能には行列サイズに対する鋸歯状の性能変動が見られる．この原因として，CUDA における典型的な GEMV 実装はスレッドブロックが行列 A の行数に比例して起動する仕組みであり，起動したスレッドブロック数が，GPU が単位時間に処理できるスレッドブロック数で割り切れなくなるときに，GPU

の実行効率が低下して性能が低下するためであると考えられる（詳しくは [15] 等で考察している）．KBLAS のシングル GPU 実装ではカーネルを 2 次元グリッドで起動して y 次元方向に複数スレッドブロック（例えば SGEMV-N の場合 8）を割り当てることで起動スレッドブロック数を増やし，性能変動の周期を小さくすることで性能を安定化しているが，マルチ GPU 版実装では y 次元方向のスレッドブロック数を GPU 数と等しくしているため，GPU 数が少ない場合には性能変動を十分に抑制できていない．一方，MUBLAS はスレッドブロックサイズを問題サイズに応じて調整し，起動スレッドブロック数を GPU のスレッドブロック処理単位にフィットさせることで，性能を安定化している．また，KBLAS では鋸歯状の性能変動以外にも細かな性能変動が見られるが，これは問題サイズが行列分割のブロックサイズに対して端数となる場合とそうでない場合で呼び出すカーネルを切り替えており，それぞれのカーネルの性能に差違があるためであると考えられる．

なお，ASPEN.K2 の SYMV の場合と同様に，MUBLAS は複数 GPU 実行時のみランダムな性能の揺れが確認できるが，この原因については現時点では明らかではないため今後調査を要する．

表 1 (補足) 実装/実験に使用した HOKUSAI Greatwave の諸性能ならびにソフトウェア環境

	Tesla K20Xm × 4
GPU Name	GK110
Compute Capability	3.5
GPU Clock (MHz)	732 (boost NA)
Multiprocessors	14
CUDA Cores	2688 (=14*192)
Memory Capacity (MByte)	6144 (GDDR5)
Memory Clock (MHz)	5200 (384bit)
Memory Bandwidth (GB/s)	250
ECC Support	Enabled (ECC on で実行)
PCI bus	PCIe 3.0×16

	Host
CPU	Intel Xeon E5-2670
CPU Core 数	12 (AVX available)
CPU Clock (GHz)	2.3
Memory Capacity (GB)	64
Linux Kernel version	2.6.32
CUDA Version	7.0
Driver Version	346.46
GNU gcc Version	4.4.7

4.4 議論

ここまでの実装と予備評価から, 以下の項目について議論を深める必要がある.

(1) マルチ GPU 化に向けたプログラミング
(2) マルチノード化への展開
(3) データフォーマットの標準化

(1). 今回の BLAS カーネルでは GPU 間での通信を除いた独立性の高い部分のみの実装を行っている. ストリームによるカーネル関数の管理部分への修正には大きな負荷を必要とはしないが, 各 GPU でデバイスに渡すデータ (ポインタ) 管理がシングル GPU 版とは異ならざるを得ない. CUBLAS-XT や MAGMA, KBLAS の様に, 配列データのポインタ渡し部分がポインタ配列のポインタに変更になるため, ホスト側でのソースの修正範囲が大きくなる. Unified Memory によって GPU, CPU 間メモリ管理が将来的に容易になることを期待したいが, 現時点では下位ライブラリで全ての組み合わせを実装していないといけないため, 汎用的な数値ライブラリのマルチ GPU 化は相当の開発コストになるのではと予想される.

(2). マルチノード化は ScaLAPACK の PBLAS の GPU オフロード版ととらえれると, PBLAS が実施しているように, ローカル演算部分に対する BLAS 演算のオフロードで十分である. しかしながら, 殆どの演算でノード間通信を必要とするので, ホスト–デバイス間通信とノード間通信の隠蔽などの技術的問題が確実に出現する. GPGPU における大規模アプリケーションでもよく指摘される個所でもあり, MVAPICH のように通信フレームワーク内部に異ノード GPU 間通信をカプセル化するなどの手法導入が必要である. もちろん, ノード内マルチ GPU 化も同時に考慮する際には先述の問題点を先に解決する必要がある.

(3). シングル GPU ではデータフォーマットに関する考慮は不要であるが, マルチ GPU ではデータ分散方式が利用者の利便性に大きく寄与するため重要な事項である. ライブラリ開発の観点からは, アルゴリズムに最適な分割方式があるため, 一つの固定フォーマットを選択せずいくつかの幅を持たせられれば性能チューニングなども行いやすい. いずれにしても, ホストとのデータ通信を考慮すると, フォーマットの標準化は避けられないであろう. 選択肢として考えられるのは, 負荷バランスが完全に均等化できるサイクリック分割, もしくは適切な固定ブロック幅 (16, 32, 48 など) によるブロックサイクリック分割であろう. さらに, 1 次元分割, 2 次元分割についても選択肢が考えられる. 現状ではシングルノード上の GPU 数は多くても 8 程度なので 1 次元分割でも十分であろう.

5. まとめ

本研究では, SYMV, GEMV のマルチ GPU 化を行った. 自動チューニング技術による最適化されたカーネルを基にしているため, 高性能かつポータブルなカーネルの構築に成功している. 先行研究である KBLAS などと比較しても高い性能と安定性を示している. 特に, GEMV は適切なス

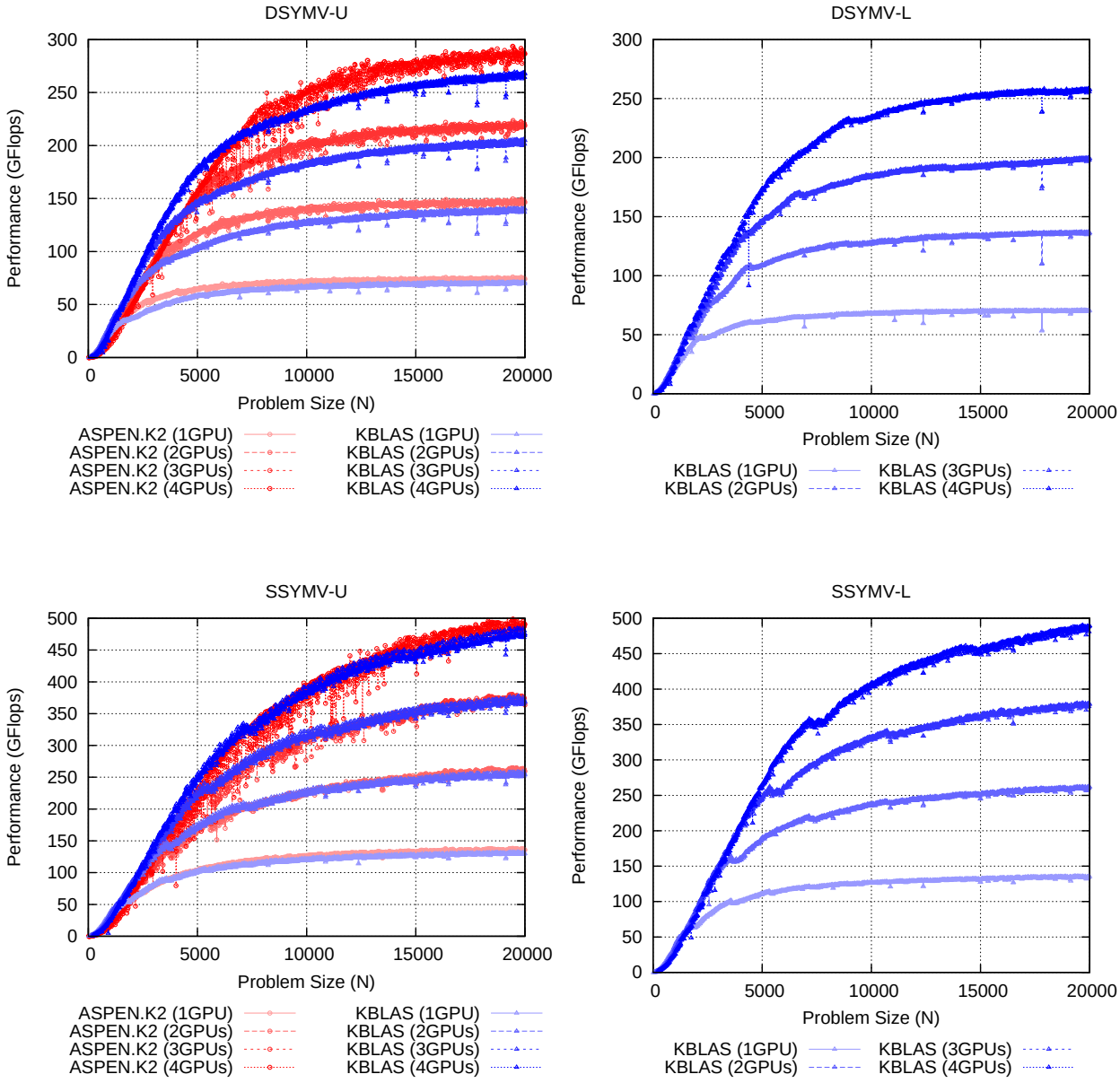

図 3 Tesla K20Xm での SYMV-(UL) の性能 (それぞれ行列は 16 次元毎に測定)

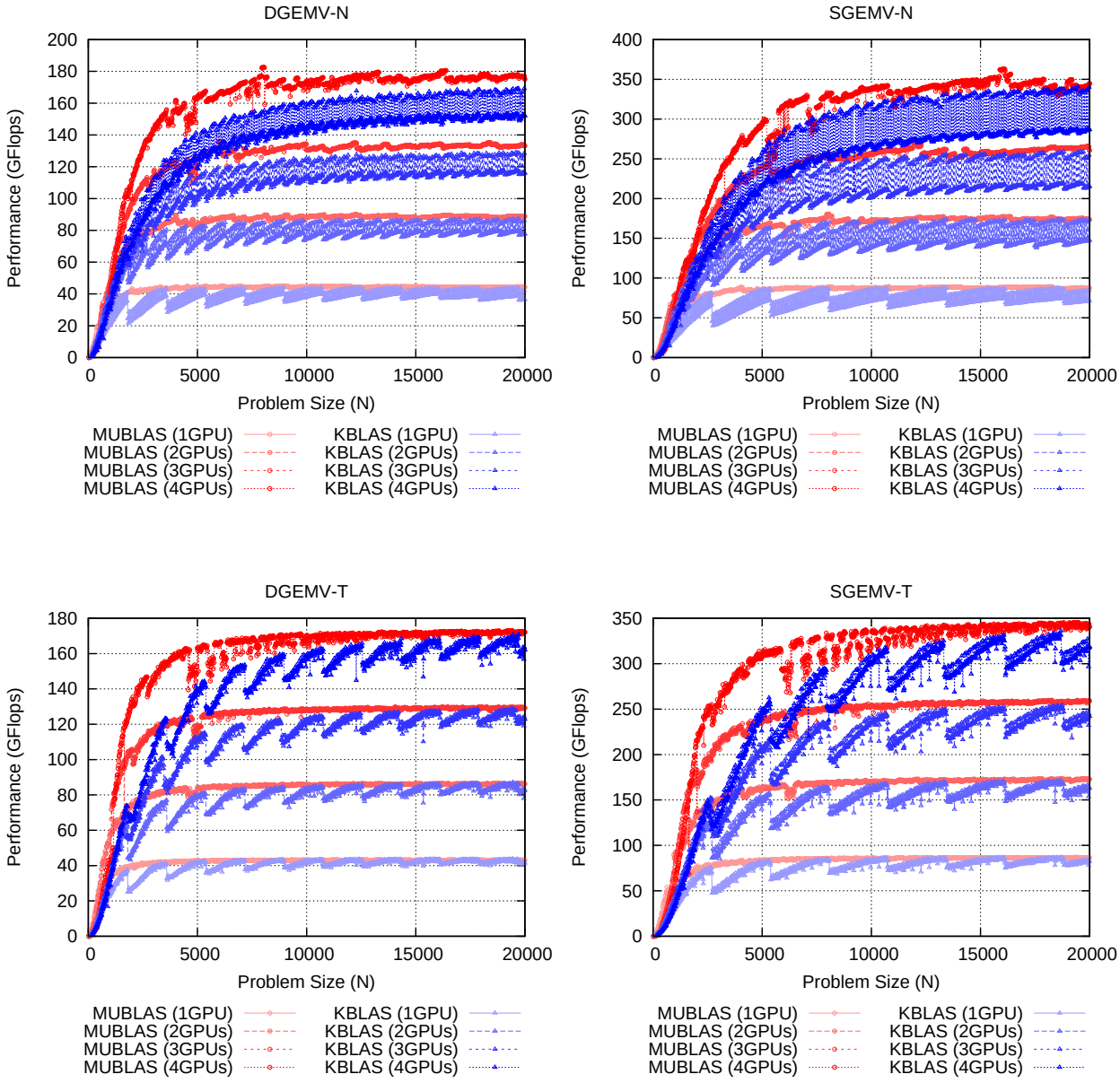

図 4 Tesla K20Xm での GEMV-(NT) の性能 (それぞれ行列は 16 次元毎に測定)

レッド形状を決定しているため鋸歯状の性能特性は見られない. 本研究は実用化に向けた予備実験の位置付けにあり, 実装技術上の課題はまだ多い. 今後, 問題点を整理し高性能・高精度なマルチ GPU 版 GPUBLAS の開発に展開していく.

本研究は科研費基盤 B(課題番号: 15H02709) ならびに計算科学振興財団 (研究助成研究教育拠点 (COE) 形成事業) の支援を受けている.

参考文献

[1] NVIDIA Corporation, CUDA C Programming guide, http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, PG-02829-001_v7.0 (2015).
[2] NVIDIA Corporation, The NVIDIA CUDA Basic Linear Algebra Subroutines, http://developer.nvidia.com/cublas
[3] Innovative Computing Laboratory, University of Tennessee, Matrix Algebra on GPU and Multicore Architectures, http://icl.cs.utk.edu/magma
[4] 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)
[5] 椋木大地, 今村俊幸, 高橋大介, Kepler アーキテクチャ GPU における高速な SGEMV の実装, GPU Technology Conference Japan 2014, (2014)
[6] 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.
[7] 今村俊幸, 内海貴弘, 山田進, 町田昌彦, CUDA-xSYMV の実装と評価, 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. 2014-HPC-146, No. 14 (2014) 1–12.
[8] 今村俊幸, 椋木大地, 山田進, 町田昌彦, CUDA-BLAS 等の選択による最速 GPU 固有値ソルバーの性能評価, 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. 2015-HPC-148, No. 4 (2015) 1–9.
[9] 椋木大地, 今村俊幸, 高橋大介, NVIDIA GPU におけるメモリ律速な BLAS カーネルのスレッド数自動選択手法, 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. 2015-HPC-150, No. 13 (2015) 1–13.
[10] NVIDIA Corporation, CUBLAS-XT – Accelerate BLAS calls with multiple GPUs!, https://developer.nvidia.com/cublasxt
[11] Abdelfattah, A., Keyes, D., and Ltaief, H., KAUST BLAS (KBLAS), http://cec.kaust.edu.sa/Pages/kblas.aspx
[12] 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, GPU Technology Conference (GTC) 2014 (2014).
[13] 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.
[14] 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.
[15] Mukunoki, D., Imamura, T., and Takahashi, D., Fast Implementation of General Matrix-Vector Multiplication (GEMV) on Kepler GPUs, 23rd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP) (2015) 642–650, doi:10.1109/PDP.2015.66.