Vol.2015-HPC-150 No.13
2015/8/4

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

椋木 大地[1,a)]　今村 俊幸[1,b)]　高橋 大介[2,c)]

概要：NVIDIA GPU のカーネルプログラムにおいてスレッド数の選択は性能に大きな影響を与えることが知られているが，最適なスレッド数を理論的に一意に決定する方法は明らかではない．本稿では性能がメモリ律速となる BLAS ルーチンである SAXPY, SGEMV, STRMV において，計算する問題サイズに対して最適なスレッド数を決定するための自動チューニング手法を検討した．提案手法は 2 つの自動チューニング機構で構成される．まず，ある問題サイズに対するカーネルのサンプリング実行から，デバイスおよびカーネル固有のチューニングパラメータを決定するオフライン自動チューニングを行う．そしてそれらのパラメータに基づいて，問題サイズに応じた最適なスレッド数を，ある性能モデルを用いてオンライン自動チューニングで決定する．評価実験では，3 つの NVIDIA GPU アーキテクチャ（Fermi，Kepler，Maxwell）において，スレッド数の選択がカーネルの性能に与える影響を示した上で，提案する手法によって多くの場合に最適なスレッド数を選択できることを示す．

1. はじめに

今日の GPU は数千レベルのコアを持つメニーコアアーキテクチャであり，その上ではプログラムが数千レベルのスレッドによるマルチスレッディングで処理される．NVIDIA 社の GPU コンピューティングプラットフォームである CUDA においては，同一の命令を複数のスレッドが実行することにより，命令レイテンシを隠蔽するという処理方式を採用し，これを Single Instruction Multiple Threads（SIMT）と呼んでいる．プログラムの性能を最大化するためには，ハードウェアリソースを可能な限り動作させることができるようなスレッド数を選択することが鍵となるが，その最適値を理論的に一意に決定する方法は NVIDIA 社のドキュメント類には示されておらず，経験的あるいは実験的にスレッド数を決めている場合が多いと考えられる．

マルチコア・メニーコアアーキテクチャにおいては，ハードウェアリソースの数の違いや計算する問題サイズによって，スレッド数などのパラメータを決定する必要がある．特に本稿が取り上げる BLAS ルーチンのような数値計算ライブラリの開発においては，任意のデバイス，任意の問題サイズにおいて動作することを想定し，同じアルゴリズムに対して演算精度が異なる複数のバリエーションを用意するため，自動チューニングによる性能最適化が効果的であると言える．また自動チューニングのための理論は，GPU 向けコードの自動生成や，古いアーキテクチャ向けに実装されたコードの改良などに役立つと期待される．

本稿では著者らが開発している NVIDIA GPU を対象とした自動チューニング機能付きの BLAS ルーチン群「MUBLAS」[1] の，GEMV-N（GEneral Matrix-Vector multiplication for Non-transposed matrix: $y = \alpha Ax + \beta y$），GEMV-T（GEMV for Transposed matrix: $y = \alpha A^T x + \beta y$），AXPY（$y = \alpha x + y$），TRMV-L（TRiangle Matrix-Vector multiplication for Lower triangler matrix: $x = Ax$）カーネルにおけるスレッド数自動選択手法について報告する．MUBLAS が対象としているハードウェアは，CUDA 7.0 が公式にサポートする Compute Capability 2.0 以上の GPU，いわゆる Fermi，Kepler，Maxwell アーキテクチャ GPU である．我々はこれまでに Kepler アーキテクチャ GPU における高速な GEMV-N カーネルの実装手法 [2] を報告しており，本稿はその発展にあたる．

MUBLAS では，GPU プログラム（カーネル）を起動する際のスレッド数の決定を，2 つの自動チューニング機構によって完全に自動化している．一つ目の自動チューニング機構は MUBLAS のビルド時に GPU のハードウェア（デバイス）とカーネル固有のパラメータをカーネルのサ

[1] 理化学研究所計算科学研究機構
[2] 筑波大学システム情報系
[a)] daichi.mukunoki@riken.jp
[b)] imamura.toshiyuki@riken.jp
[c)] daisuke@cs.tsukuba.ac.jp

ンプリング実行により決定する．二つ目の自動チューニング機構は，BLAS ルーチンが計算する問題サイズに応じてハードウェアの実行効率を最大化するようなスレッド数を理論的に決定する．自動チューニングの種類をオンラインとオフラインに分類すると [3]，前者は試行によってパラメータを決定するオフライン自動チューニングであり，後者は試行を必要とせずユーザがルーチンをコールするたびに内部的にチューニングが行われるオンライン自動チューニングである．

NVIDIA GPU 向けの BLAS 実装としては, NVIDIA による CUBLAS[4] や，テネシー大 ICL の開発する MAGMA[5] に含まれる MAGMA BLAS が知られている．このほか KBLAS[6][7] は GEMV・SYMV・TRMM のみであるが最適化された実装を公開している．GPU における BLAS ルーチンの自動チューニングとしては，GEMV 等のメモリ律速なルーチンに対する研究 [8] [9] [10] や，GEMM[11] の事例が報告されている．また ASPEN.K2[12] は自動チューニングによる GEMV，SYMV を提供するオープンソースソフトウェアであり，ユーザ環境において自動チューニングを行うためのフレームワークも公開されている．これらの事例においてスレッド数の選択はチューニングパラメータの一つとなっているが，いずれもオフライン自動チューニングによって実験的に最適値を決定している．一方，我々の研究は可能な限り理論に基づくオンライン自動チューニングを取り入れることを主眼としている．

なお，本稿の内容は MUBLAS 1.4.28 に基づくものであり，実験に用いたソースコードはオープンソースソフトウェアとして公開している [1].

2. CUDA のマルチスレッディングと最適なスレッド数の選択

本章では CUDA のマルチスレッディングの仕組みに基づいて，CUDA プログラムのスレッド数を決定するための基本的な方針と，いくつかの指標に基づく最適なスレッド数の決定手法について説明する．なお，**図 1** は本章の内容を補足するものである．

2.1 CUDA のアーキテクチャとマルチスレッディング

まず予備知識として，CUDA におけるメニーコアアーキテクチャとマルチスレッディングの仕組みを簡単に述べる．GPU には数千レベルのコア（CUDA Core）が搭載されている．CUDA Core は複数個が，マルチプロセッサと呼ばれるユニットにまとめられており，マルチプロセッサにはスレッドのスケジューラ，命令発行ユニット，Load/Store ユニット，キャッシュあるいはスクラッチパッドメモリとして利用可能なオンチップメモリが搭載されている．

CUDA では 1 つのカーネル（GPU で実行される関数）が複数（通常数百から数千）のスレッドとして実行される．スレッドはスレッドブロックという単位でマルチプロセッサに割り当てられ，マルチプロセッサはスレッドブロック内のスレッドをワープと呼ばれる 32 スレッド単位でスケジューリングし，実行していく．1 つのスレッドは 1 つの CUDA Core で実行される．マルチプロセッサは非常に高速なコンテキストスイッチを持ち，同一の命令を複数のワープ（スレッド）が並列に実行することにより，レイテンシを隠蔽する．すなわち，複数のワープが実行状態にあり，そのうちのあるワープがレイテンシにより実行待ち（ストール）の状態にある時に，他のワープを実行する．同一マルチプロセッサ内のスレッドはバリア同期を行ったり，共有メモリを介してデータを交換することができる．スレッドブロックの集合体をグリッドと呼び，1 つのカーネルは 1 つのグリッドに対応する．スレッドおよびスレッドブロックはそれぞれ最大 3 次元（x, y, z）の構造を持つことができ，プログラミング言語からはそれらの ID を得ることができる．

2.2 設定可能なスレッド数

スレッド数は 1 以上の任意の整数を設定できるが, CUDA の仕様 [13] によりスレッド数およびスレッドブロック数に上限がある．1 グリッドにおけるスレッドブロック数の上限は x 次元方向が $2^{31}-1$（Compute Capability 2.x の GPU のみ 65535），y 次元および z 次元は 65535 である．また 1 スレッドブロックにおけるスレッド数の上限は x 次元および y 次元が 1024，z 次元が 64 であるが，すべての次元をあわせたトータルで 1024 スレッドという上限がある．なお，本稿の課題である最適なスレッド数を決定することとは，例えばスレッドブロックが 2 次元で構成されている場合には，最適なスレッドブロックサイズ（x 次元のスレッド数, y 次元のスレッド数）の組み合わせを見つけることである．

カーネルを構成するグリッドはスレッドブロック単位でマルチプロセッサに割り当てられ，スレッドブロック内のスレッドが同時に実行していく．このときスレッドブロック内のスレッドは，マルチプロセッサが持つリソース（レジスタと共有メモリ）を取り合って使用する．そのため 1 スレッドが使用するリソース量によって，1 スレッドブロックあたりのスレッド数に制約が加わる．例えば 1 マルチプロセッサに 65536 個のレジスタがあり，1 スレッドがレジスタを 128 個使用するカーネルでは，65536/128=512 スレッドが上限となる．1 スレッドブロックが使用するリソース量がマルチプロセッサのリソース量を上回るようなスレッド数を指定してカーネルを起動しようとするとエラーが発生する．あるカーネルを起動する際に指定できるスレッドブロックあたりのスレッド数の最大値は CUDA の API 関数 `cudaFuncGetAttributes()` を用いて知ることができる．

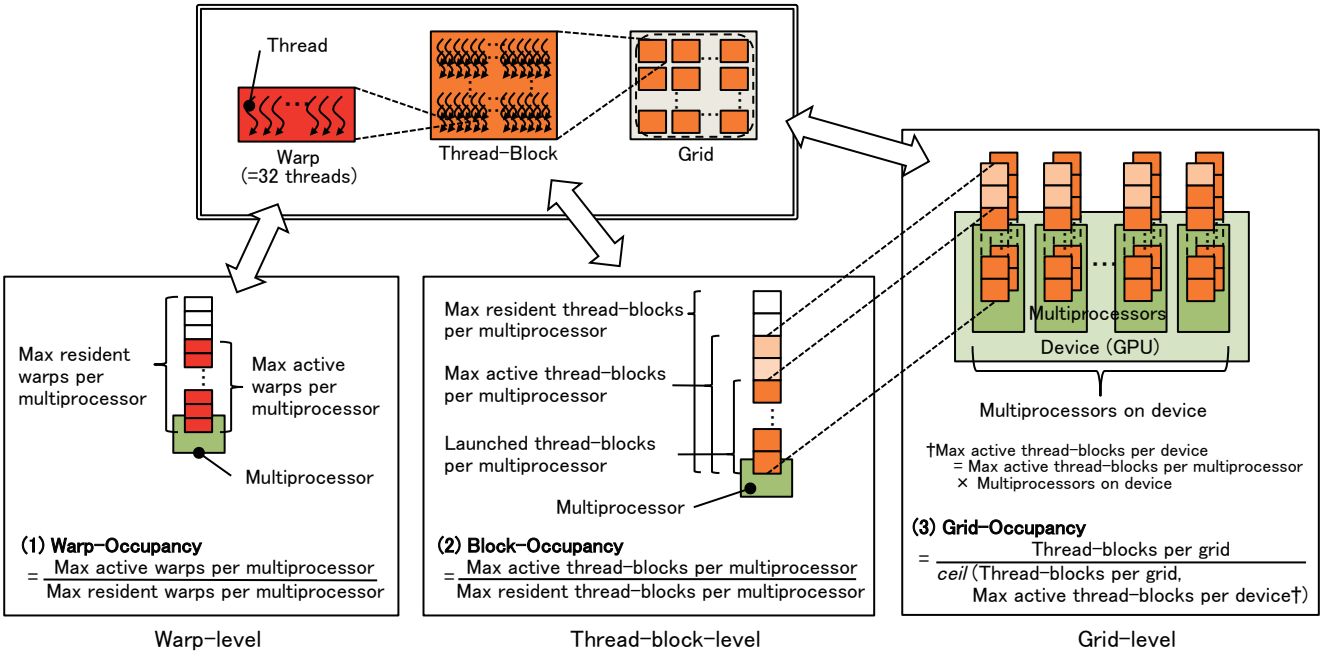


図 1 スレッド階層と 3 つの Occupancy

なお，カーネルの設計によっては取りうるスレッド数に制約が加わる場合がある．例えばワープシャッフル命令を使用したコードなど，ある固有のスレッド数でなければ動作しないような実装を行った場合である．

2.3 最適スレッド数決定のための条件と指標

2.3.1 基本条件

スレッド数は，マルチプロセッサがワープ（32 スレッド）単位でスレッドを実行することから，32 の倍数で設定することが多い．また，CUDA におけるメモリアクセスの特徴として，連続するスレッドによる 128Bytes 境界にアラインメントされた 128Bytes の連続領域に対する同時アクセスが，1 メモリトランザクションとして実行される．1 スレッドは 1 命令で 4/8/16Bytes のデータにアクセスできるため，例えば各スレッドが 8Bytes アクセスを行った場合は，128/8 = 16 スレッドずつメモリトランザクションが実行される．したがって，性能がメモリアクセスに依存するカーネルでは，連続するスレッド数がメモリアクセスを行うスレッド数単位で設定されていることが望ましい．

スレッドまたはスレッドブロックが 2 次元あるいは 3 次元で設定されている場合には，カーネルの設計上，どの次元を大きくあるいは小さく設定した方が良いというプライオリティをつけられる場合がある．例えば，ある次元のスレッド数に依存してキャッシュや共有メモリへのブロッキング効果が大きくなるような場合は，その次元のスレッド数はなるべく大きく設定するべきだと言える．

また，スレッド数の選択においては，マルチプロセッサのリソースを無駄なく使用するために必要な最小スレッド数が存在すると考えられるが，著者らの知る限りでは最小スレッド数を決定するための理論は明らかではない．仮に浮動小数点演算命令のみを実行するようなカーネルであれば，マルチプロセッサあたりの CUDA Core 数に相当するスレッド数が最小スレッド数となることも考えられるが，実際の多くのカーネルでは，CUDA Core 数以下のスレッド数であっても十分な性能が得られるケースがある．これはメモリアクセスや同期などさまざまな命令と，Load/Store ユニットやワープスケジューラの数が関係しているものと推測できる．

2.3.2 Warp-Occupancy

Warp-Occupancy は最適なスレッド数を決定するための指標の一つとなる概念である．

マルチプロセッサは同一の命令を複数のワープが並列に実行することによりレイテンシを隠蔽する．すなわち，複数のワープが実行状態にあり，そのうちのあるワープがレイテンシにより実行待ち（ストール）の状態にある時に，他のワープを実行する．したがって，カーネルの実行性能は，(a) カーネルのワープストールの発生具合，(b) ワープストールをカバーするために必要なワープ並列数，(c) 同時に実行できるワープ数，によって決まる．(a) は実行バイナリに依存，(b) はハードウェアに依存，(c) はハードウェアと実行バイナリおよび実行スレッド数に依存する．そして (b) が (c) を上回る状態にあるときにカーネルの性能は低下する．

(c) は，(i) ハードウェアの仕様上の上限ワープ数（Max resident warps per multiprocessor：$MaxRsdWrpPerMp$）と，(ii) カーネルの実装と実行パラメータによって決まる同時実行可能ワープ数（Max active warps per multiprocessor：$MaxActvWrpPerMp$）によって決まる．(i) は Fermi

アーキテクチャで 48 ワープ，Kepler・Maxwell アーキテクチャで 64 ワープである．(ii) については，マルチプロセッサ内で並列に実行されるワープがマルチプロセッサが持つリソース（レジスタ・共有メモリ）を共有するため，1 つのワープ（スレッド）が要求するリソース量とマルチプロセッサあたりのリソース量，およびバイナリの実行スレッド数（ワープ数）によって，並列に実行できるワープ数が制限されることがある．例えば，1 つのマルチプロセッサには 65536 個の 32bit レジスタがあるが，1 スレッドが 64 個のレジスタを使用する場合，1 ワープあたり 2048 個のレジスタを使用するため，65536/2048 = 32 個しかワープを同時実行できない．

(c) に関して NVIDIA は，(i) に対する (ii) の割合を Occupancy（占有率）として定義しており [14]，本稿ではこれを Warp-Occupancy（$WrpOcp$）と呼ぶ．すなわち，

$$WrpOcp = MaxActvWrpPerMp/MaxRsdWrpPerMp$$

である．Warp-Occupancy は実行バイナリが達成可能なワープレベルの並列性を示したものであり，カーネルのチューニングの指標の一つとして知られている．しかし，ワープストールが発生しにくいカーネル，すなわち (a) が小さい場合には，Warp-Occupancy が低くても高い性能が得られる場合がある．

2.3.3 Block-Occupancy

最適なスレッド数を決定するためのもう一つの指標となりうる Block-Occupancy について説明する．

マルチプロセッサにおいて複数のワープが同時に実行されていることと同様に，マルチプロセッサは複数のスレッドブロックを同時に実行することで，スレッドブロックのストールを隠蔽している．したがって，スレッドブロックにおいても Warp-Occupancy に関する (a)–(c) および (c) の (i) と (ii) に相当する概念が存在する．スレッドブロックのストールは主にスレッドブロック内のスレッド同期処理によって生じると考えられる [14]．

(c) の (i) に相当する概念として，ハードウェアの仕様上の上限スレッドブロック数（Max resident thread-blocks per multiprocessor：$MaxResdBlkPerMp$）がある．この値は Fermi アーキテクチャ：8，Kepler アーキテクチャ：16，Maxwell アーキテクチャ：32 である．また (c) の (ii) に相当する概念として，カーネルの実装と実行パラメータによって決まる同時実行可能スレッドブロック数 (Max active thread-blocks per multiprocessor：$MaxActvBlkPerMp$) が，ワープと同様に 1 スレッドあたりのリソース使用量とマルチプロセッサのリソース量から決まる．

NVIDIA の文献にはスレッドブロックレベルの並列性を示す指標は定義されていないが，本研究では Warp-Occupancy の概念をスレッドブロックに対して適用し，Block-Occupancy（$BlkOcp$）として以下のように定義する．

$$BlkOcp = MaxActvBlkPerMp/MaxResdBlkPerMp$$

2.3.4 Grid-Occupancy

グリッドあたりの最適なスレッドブロック数を決定するための指標となる Grid-Occupancy[*1] について説明する．

1 つの GPU（CUDA ではデバイスと呼ぶ）には複数のマルチプロセッサが搭載されており，グリッドは複数のスレッドブロックとして複数のマルチプロセッサ上で実行される．したがって，すべてのマルチプロセッサ数を均等に稼働させるためには，スレッドブロック数はマルチプロセッサの倍数であることが望ましいと言える．一方，Block-Occupancy の説明で述べたように，1 つのマルチプロセッサは複数のスレッドブロックを同時に実行できる．したがって，1 グリッドあたりの理想的なスレッドブロック数は，マルチプロセッサあたりの同時実行可能スレッドブロック数（$MaxActvBlkPerMp$）にマルチプロセッサ数を掛けた値である．これをデバイスあたりの最大同時実行可能スレッドブロック数（Max active thread-blocks per device：$MaxActvBlkPerDev$）とする．

カーネルのグリッドあたりのスレッドブロック数（$BlkPerGrd$）と理想的なスレッドブロック数の関係を，Grid-Occupancy（$GrdOcp$）として次のように定義する．

$$GrdOcp = BlkPerGrd/\texttt{ceil}(BlkPerGrd, MaxActvBlkPerDev)$$

`ceil` (x, y) は x を最近接の y の倍数へ切り上げた数を返す．グリッドあたりのスレッドブロック数がデバイスあたりの同時実行可能スレッドブロック数の倍数である時に Grid-Occupancy は 100%となり，GPU は最大効率で動作すると考えられる．

2.4 Occupancy に基づく最適なスレッド数の決定

最適なスレッド数の選択に関しては NVIDIA の文献 [14] の 10.5 節にいくつかの経験則が示されているが，理論的な決定方法は示されていない．本研究ではこれまでに本章で示したスレッド数選択の基本条件といくつかの指標から，カーネルの性能を最大化するための最適なスレッド数は，以下のようにして決定できると考える．

(1) スレッド数がワープサイズまたはメモリアクセス単位の倍数
(2) Grid-Occupancy が可能な限り大きい
(3) 必要十分な Warp-Occupancy がある

[*1] 我々の先行研究 [2] では Grid-Occupancy を Block-Occupancy と呼んでいたが，Warp-Occupancy の定義をスレッドブロックレベルに自然に拡張すると本稿の Block-Occupancy という定義ができるため，呼称を改めた．

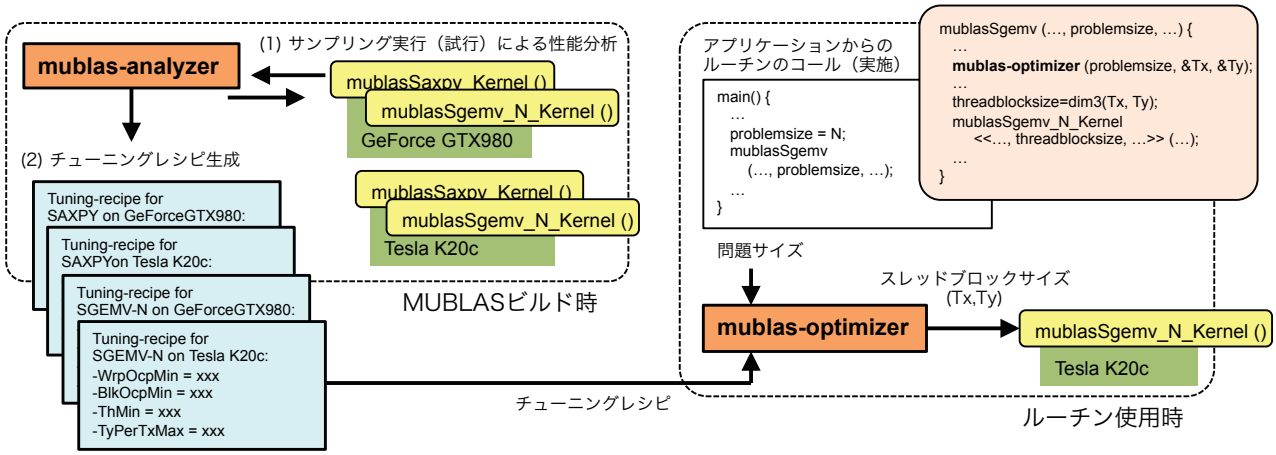


図 2 MUBLAS の概要

(4) 必要十分な Block-Occupancy がある
(5) ある一定量以上のスレッド数がある
(6) カーネル設計上の条件を満たす

これらの要素の中で，(1) と (2) を満たすようなスレッド数は，デバイスおよびカーネルに依存せずに理論的に決定できる．しかし (3)–(5) の最小スレッド数，最小 Warp-Occupancy，最小 Block-Occupancy は，デバイスとカーネルに依存して異なる可能性がある．また (6) はカーネルの実装に依存する．そのため，(3)–(6) については，デバイス，カーネルごとに何らかのモデルを構築して理論的に与えるか，実験的に求める必要がある．

また，3 つの Occupancy のうち Grid-Occupancy は，Warp-Occupancy と Block-Occupancy の 2 つと異なる性質を持つ．前者の 2 つは，1 つのマルチプロセッサにおけるワープレベルもしくはスレッドブロックレベルの並列度を示している．そしてレイテンシを隠蔽するために必要なこれらの並列度は，ハードウェアとプログラムに対して固有の必要十分な値をとると考えられる．すなわち必ずしも 100%を達成する必要はない．一方で Grid-Occupancy はハードウェアの稼働率であり，グリッド内のスレッドブロックがデバイス上のマルチプロセッサをどれだけ占有しているかを表している．したがって，Grid-Occupancy は 100%を達成することが望ましいと言える．Grid-Occupancy はスレッドブロック数に依存するため，BLAS ルーチンなどで問題サイズに比例してスレッドブロックが起動するような実装を行っている場合には，問題サイズごとに最適なスレッド数が異なる．そのためスレッド数を固定して問題サイズに対する性能を測定すると，鋸歯状の周期的な性能変動がみられることがある [8][2].

3. MUBLAS の実装

MUBLAS は，前節に示した最適なスレッド数選択の理論に基づいてスレッド数を決定する．本章では前半でその自動チューニング機構の実装について，後半では 4 つの BLAS カーネル（AXPY，GEMV-N，GEMV-T，TRMV-L）の実装について示す．

3.1 自動チューニング機構の実装

MUBLAS では 2.4 節に示した Occupancy に基づくスレッド数の決定を，2 つの自動チューニング機構によって実現している（**図 2**）．一つ目がデバイスとカーネル固有のチューニングパラメータを決定するためのオフライン自動チューニングであり，二つ目が問題サイズによって最適なスレッド数を自動で決定するオンライン自動チューニングである．それぞれの詳細を次に示す．

なお，MUBLAS のカーネルはグリッドが 1 次元，スレッドブロックは 2 次元までの構造を用いており，自動チューニング機構は現時点ではそれ以上の次元数を考慮していない．また，以降では 2 次元のスレッドブロックにおいて x 次元のスレッド数を Tx，y 次元のスレッド数を Ty と呼ぶ．

3.1.1 デバイスとカーネル固有のチューニングパラメータの決定

まず 2.4 節に示した (3)–(6) について，カーネルのサンプリング実行によってパラメータを決定する．このプロセスを「mublas-analyzer」と呼び，MUBLAS のビルド時に実行されるオフライン自動チューニングとなっている．mublas-analyzer が決定するパラメータは以下の 4 つである．

- Warp-Occupancy の最小値（$WrpOcpMin$）
- Block-Occupancy の最小値（$BlkOcpMin$）
- スレッド数の最小値（$ThMin$）
- Tx と Ty の比率 Ty/Tx の最大値（$TyPerTxMax$）

これらの最大値・最小値は，カーネルがある目標性能を達成するために必要な最小値・最大値を意味する．これらはデバイスとカーネル固有のパラメータであり，デバイス，カーネルごとに取得する必要がある．MUBLAS ではこれらの 4 つのパラメータを，デバイスとカーネルごとの「チューニングレシピ」として扱う．なお，4 番目のパラ

メータ $TyPerTxMax$ は 2.4 節 (6) の「カーネル設計上の条件」に相当するもので，スレッドブロックを 2 次元で起動する GEMV-N と TRMV-L カーネルで用いる．これらのカーネルは Tx に比例して共有メモリブロッキングあるいはキャッシュの効果が向上する．一方で Ty に比例して総和計算の実行時間が増加する．したがって，Ty/Tx はなるべく小さくすべきであり，その許容できる最大値をパラメータとして用いる．

4 つのパラメータは次のようにして決定される．まず，MUBLAS のカーネルは任意のスレッドブロックサイズで起動できるように実装されており，さらにカーネル実行時の Warp-Occupancy，Block-Occupancy を MUBLAS のハンドラ経由で取得することができる．そこで任意のスレッドブロックサイズでカーネルを起動し，性能と Warp-Occupancy，Block-Occupancy，Th，$TyPerTx$ を表示するカーネル評価プログラムを用意する．そして Python スクリプトで選択候補となるすべてのスレッドブロックサイズでカーネルを実行（サンプリング実行）し，性能（実行時間）と，Warp-Occupancy，Block-Occupancy，Th ($= Tx \times Ty$)，$TyPerTx$ を取得する．このサンプリング実行はカーネルの性能が十分に発揮できるような適当な問題サイズで実行し，この実行で得られた最大性能の 98%を目標性能と定める．そして最大性能の 98%以上の性能を達成した最小の Warp-Occupancy を $WrpOcpMin$ とする．他の 3 つのパラメータについても同様にして決定する．決定された 4 つのパラメータはチューニングレシピとしてファイルに出力され，後述する mublas-optimizer のスレッド数選択時に考慮される．

なお現時点では「適当な問題サイズ」と「目標性能=最大性能の 98%」の決め方について検討の余地が残っている．ただし「適当な問題サイズ」は大きなスレッド数であっても十分な性能が得られるような，ある程度大きな問題サイズを用いるべきであると言える．また目標性能をタイトに設定しすぎると，チューニングレシピの最小値・最大値を満たす候補が見つからなくなる可能性が出てくる．

3.1.2 問題サイズに対する最適なスレッド数の自動決定

問題サイズに対する最適なスレッド数の自動決定は，2.4 節に示した (1) の条件および (2) の Grid-Occupancy に基づいて行われる．このプロセスを「mublas-optimizer」と呼ぶ．mublas-optimizer は Grid-Occupancy と前述の過程で生成したチューニングレシピに基づいて，問題サイズに対する最適なスレッドブロックサイズを決定する．

mublas-optimizer によるスレッド数選択プロセスはルーチンがコールされる度にカーネル実行前に CPU 側で実行される．したがって，ユーザから見てオンライン自動チューニングとして動作する．このプロセスは選択候補となるすべてのスレッドブロックサイズについて 3 つの Occupancy を計算し，最も条件が良いスレッドブロックサイズを返す．**図 3** に最適なスレッドブロックサイズ決定のためのアルゴリズムの疑似コードを示す．このアルゴリズムは 2 次元スレッドブロック用であるが，1 次元の場合は $TyMin = TyMax = 1$ とする．アルゴリズムの補足を以下に記す．

```
1  if (TxStep == 0)
2    TxStep = 128 / (sizeof(TYPE) * Nt);
3  if (ThMin == 0)
4    ThMin = TxStep;
5  TxMin = ThMin;
6  TxMax = ThMax = funcatb.maxThreadsPerBlock;
7  WrpOcpTmp = 0;
8  BlkOcpTmp = 0;
9  GrdOcpTmp = 0;
10 for (Tx=TxMax; Tx>= TxMin; Tx−=TxStep){
11   TyMin = ThMin / Tx;
12   TyMax = max (1, (ThMax / Tx));
13   if (TyPerTxMax != 0)
14     TyMax = min (TyMax, TyPerTxMax*Tx);
15   for (Ty=TyMin; Ty<= TyMax; Ty++){
16     GetOcp (Tx,Ty,&WrpOcp,&BlkOcp,&GrdOcp);
17     if ( GrdOcp > GrdOcpTmp
18     && WrpOcp > min (WrpOcpTmp, WrpOcpMin)
19     && BlkOcp > min (BlkOcpTmp, BlkOcpMin)){
20       WrpOcpTmp = WrpOcp;
21       BlkOcpTmp = BlkOcp;
22       GrdOcpTmp = GrdOcp;
23       TxOptimal = Tx;
24       TyOptimal = Ty;
25     }
26   }
27 }
```

図 3 mublas-optimizer における最適なスレッドブロックサイズを決定するためのアルゴリズム

- x 次元のスレッド数は $TxMin$ から $TxMax$ の間で $TxStep$ の倍数から選び，$TxStep$ は指定がない限りはメモリアクセス単位のスレッドを設定する（1 行目において単精度の場合 TYPE=float，$Nt = 4$ である）
- スレッドブロックあたりのスレッド数 ($Th = Tx \times Ty$) は $ThMin$ から $TxMax$ とし，チューニングレシピで $ThMin$ の指定がない場合は，$ThMin = TxStep$ とする
- 6 行目の funcatb.maxThreadsPerBlock は CUDA API 関数の `cudaFuncGetAttributes()` で取得したカーネルを起動できる最大スレッド数である
- 16 行目の GetOcp (Tx, Ty, ...) は Tx, Ty に対する 3 つの Occupancy（$WrpOcp$，$BlkOcp$，$GrdOcp$）を計算する
- 条件を満たす候補が複数存在する場合は，なるべく x 次元（メモリのアドレス連続方向）のスレッド数が大きく，y 次元のスレッド数が小さくなるような候補を

選ぶ（この条件は $TyPerTxMax$ にちなむ）

まとめると，このアルゴリズムで選択される「最も条件が良いスレッドブロックサイズ」とは，「(1) スレッド数がメモリアクセス単位の倍数で，(2) Grid-Occupancy が可能な限り大きく，(3) $WrpOcpMin$ 以上の Warp-Occupancy があり，(4) $BlkOcpMin$ 以上の Block-Occupancy があり，(5) $ThMin$ 以上のスレッド数で，(6) Tx はなるべく大きく Ty が $Tx \times TyPerTxMax$ 以下でなるべく小さい」スレッドブロックサイズである（括弧付き番号は 2.4 節の (1)–(6) に対応する）．

3 つの Occupancy の計算方法は Warp-Occupancy の計算を基本として大部分が共通である．我々は CUDA Toolkit に含まれている Warp-Occupancy を計算するためのエクセルシートである CUDA Occupancy Calculator を参考にして，計算式を実装した．3 つの Occupancy の計算に必要となる，カーネルのスレッドあたりのレジスタ使用量は `cudaFuncGetAttributes()`，GPU のデバイス情報（Compute Capability やマルチプロセッサ数などは `cudaGetDeviceProperties()` という API 関数で取得できる．ただし後者の関数は実行に時間を要するため，CUBLAS の cublasCreate に相当する初期化関数（mublasCreate）の中で最初に実行して値を取得し，テーブルに格納している．

3.2 BLAS カーネルの実装

SAXPY, SGEMV-N（trans=N），SGEMV-T（trans=T），STRMV-LNN（uplo=L, trans=N, diag=N）の 4 つのカーネル実装について説明する．各ルーチンの実装に共通の特徴として，グローバルメモリからのロードには 128-bit 幅のベクトルロード命令を使用し，1 スレッドが 4 要素を一度にアクセスする．この要素数を $Nt = 4$ とする．また，Fermi アーキテクチャ以降の GPU では 64KB のオンチップメモリを L1 キャッシュと共有メモリで共有し，その比率を設定できるが，SAXPY を除いて共有メモリに優先的に割り当てる `cudaFuncCachePreferShared` を設定している．

3.2.1 AXPY

AXPY は 1 次元のグリッドおよびスレッドブロックで計算する．**図 4** にカーネルの概要を図示する．長さ N のベクトルに対して ceil $(N/(Tx \times Nt), 1)$ 個のスレッドブロックを起動する．

3.2.2 GEMV-N

GEMV-N カーネルの概要を**図 5** に示す．スレッドブロックは (Tx, Ty) の 2 次元で起動する．グリッドは 1 次元の構成であり，行列 A の行数 M に比例して ceil $(M/(Tx \times Nt), 1)$ 個のスレッドブロックが起動する．行列の 1 行および結果のベクトル y の 1 要素は Ty 個のスレッドで並列に計算されるため，最後に Ty スレッド間での総和計算が発生する．この部分は y 方向のスレッド ID が 0 番のスレッドが，共有メモリを介してシリアルに総和を計算する．したがって，Ty が大きいほどこの部分のコストは大きくなる．ベクトル x は $Tx \times Ty$ スレッドで共有メモリに格納することで共有メモリブロッキングを行う．これにより 1 スレッドブロック内でベクトル x へのアクセス回数が $1/Tx$ となる．このほか行列 A のロードには Read Only Data Cache を適用した．

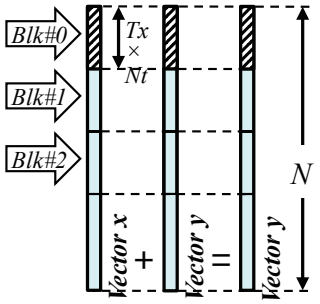


図 4 AXPY カーネル

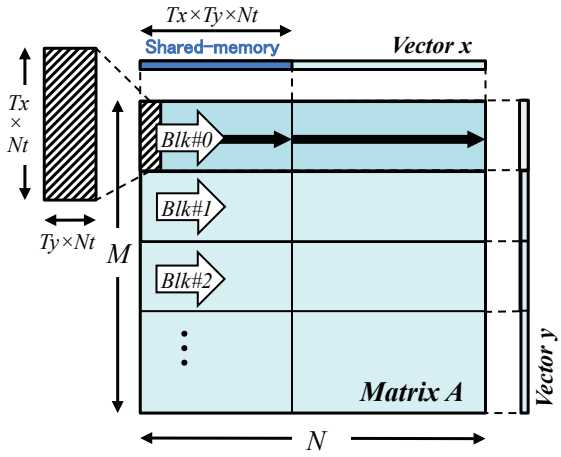


図 5 GEMV-N カーネル

3.2.3 GEMV-T

GEMV-T カーネルの概要を**図 6** に示す．GEMV-T では行方向に連続メモリアクセスとなり，CSR 形式の疎行列ベクトル積カーネルと似たメモリアクセスパターンをもつ．スレッドブロックは 1 次元で起動し，1 行あたり 1 ワープ (32 スレッド) ずつ割り当て，1 ワープ以上のスレッドは次の行を計算する．すなわち 1 スレッドブロックが $Tx/(Nt \times 32)$ 行を計算する．このためスレッド数は 32 の倍数という制約がある．グリッドも 1 次元で構成し，行列 A の行数 M に比例して ceil $(M/(Tx/(Nt \times 32)), 1)$ 個のスレッドブロックが起動する．行方向に連続となるスレッド間で総和を計算する際に，Kepler アーキテクチャ以降では warp-shuffle 命令を用いる．Fermi アーキテクチャでは共有メモリを介して計算する．ベクトル x は共有メモリ経由でアクセスし，行列 A に対して Read Only Data Cache を適用した．

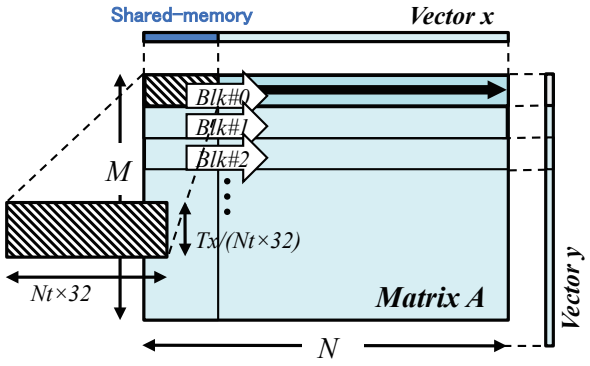


図 6 GEMV-T カーネル

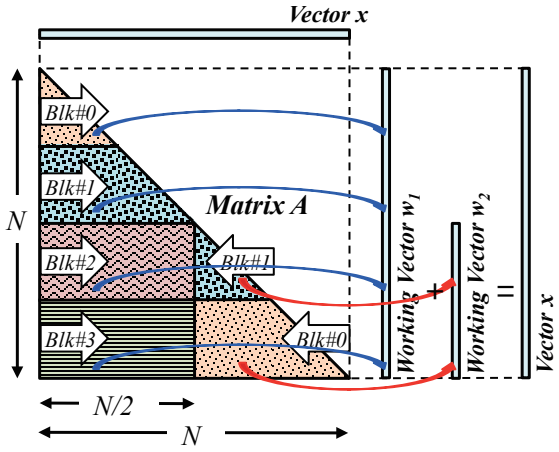


図 7 TRMV-LNN カーネル

3.2.4 TRMV-LNN

TRMV-LNN カーネルの基本的な実装は GEMV-N と同じであるが，(1) 三角行列を計算するためにメモリアクセス範囲が対角要素までとなる点，(2) 計算結果は入力に用いられたベクトル x に上書きされる点，の 2 つが GEMV-N と異なる点である．(1) の三角行列の計算については，GEMV-N の実装をベースにメモリアクセス範囲を制限するだけの単純な実装を行うと，各スレッドブロックが計算するエリアの面積が異なるため，スレッドブロック間の負荷の不均衡が生じる．そこで各スレッドブロックの負荷がおおむね均等になるようにするために，行列の上半分を計算するスレッドブロックが，行列の右下半分も計算するようにした（**図 7**）．これで各スレッドブロックの負荷はおおよそ行方向に $N/2$ で均等になる．一方，行列下半分ではベクトル x に書き戻される計算結果が 2 個のスレッドブロックで計算されることになり，それらの結果をマージする必要がある．そのためには，アトミック加算を用いる方法と，各々のスレッドブロック用にワーク領域のベクトルを用意してスレッドブロックごとに中間結果を書き出し，最後にそれらを足し合わせる方法があるが，アトミック加算を用いる方法は計算順序を保証できず計算結果の再現性が保証できないという問題があるため，我々は後者の方法で実装した．ワーク領域同士のベクトル和は別カーネルを起動して行い，このワーク領域は MUBLAS の初期化関数である mublasCreate において確保するものとする．なお，このベクトル和を行うカーネルは AXPY と同様の実装であるが，TRMV のメインカーネルと比べると実行時間が短いため，スレッド数の自動調節機構を適用せず，64 スレッドに固定して起動している．また，TRMV のメインカーネルにおいてベクトル x は GEMV-N と同様に共有メモリ経由でアクセスするべきであるが，現時点では実装が煩雑になるため共有メモリを使用していない．その代わりに行列 A とベクトル x の両方に対して Read Only Data Cache を適用した．

4. 評価実験

評価実験では MUBLAS の SAXPY，SGEMV-N，SGEMV-T，STRMV-LNN の 4 カーネルに対して，MUBLAS が対象とする 3 つの GPU アーキテクチャ（Fermi，Kepler，Maxwell）において，以下の項目を検討する．

- 選択されたスレッドの妥当性
- スレッド数の選択が性能に与える影響
- オンライン自動チューニングのコスト
- CUBLAS との比較（参考）

実験に用いた GPU を**表 1** に，ホストマシンを**表 2** に示す．カーネルのコンパイルでは nvcc にオプション”-gencode arch=compute_XY,code=sm_XY”を指定して各アーキテクチャ向けのバイナリを生成した．ここで”XY”は対象となる GPU のアーキテクチャの種類に相当する Compute Capability の数字が入る．性能はメモリ律速であるためメモリスループット（GB/s）で示し，問題サイズは SAXPY の場合 10240 の倍数で 204800 まで，その他のルーチンは 256 の倍数で 8192 までの正方行列について測定する．

4.1 チューニングレシピの生成

チューニングレシピを生成するためのサンプリング実行は，問題サイズを SAXPY で N=409600，他のルーチンは N=8192 の正方行列で行った．**表 3** に生成されたチューニングレシピの内容を示す．この結果から，デバイスあるいはアーキテクチャとルーチンによってパラメータが異なることがわかる．例えば，Tesla K20c は SGEMV-N，STRMV-LNN において Block-Occupancy が性能に影響しているが，他のデバイスでは Block-Occupancy を考慮する必要はないと言える．

表 1 実験環境（GPU）

GPU	GeForce GTX580	Tesla K20c	GeForce GTX980
Architecture	Fermi/GF110	Kepler/GK110	Maxwell/GM204
Compute Capability	2.0	3.5	5.2
CUDA Cores×Multiprocessors	32 × 16	192 × 13	128 × 16
Flops (single-precision)	1603.6 GFlops	3521.9 GFlops	4978.7 GFlops
Memory Capacity	3.2 GB	5.0 GB	4.3 GB
Memory Bandwidth	193.0 GB/s	208.0 GB/s	224.3 GB/s
ECC	N/A	Enabled	N/A
Host machine	Machine A	Machine B	Machine A

表 2 実験環境（ホストマシン）

Host Machine	Machine A	Machine B
CPU	Core i7-4790	Core i7-3930K
OS	CentOS 7.1.1503	Fedora release 18
	(3.10.0-229.4.2.el7.x86_64)	(3.11.10-100.fc18.x86_64)
CUDA	7.0	7.0
GPU Driver	352.21	346.47
Compiler	gcc 4.8.3, nvcc V7.0.27	gcc 4.7.2, nvcc V7.0.27

4.2 スレッドブロックサイズの候補数

表 4 に mublas-optimizer が候補としたスレッドブロックサイズの候補数を示す．「想定上の最大」とは，図 3 のアルゴリズムにおいて $ThMin = 0$，$ThMax$=funcatb.maxThreadsPerBlock=1024 の場合の候補数である．これに対して，チューニングレシピの $ThMin$ が 0 以外の数となっているか，funcatb.maxThreadsPerBlock が 1024 未満の数である場合には，実際の候補数はより小さくなる．funcatb.maxThreadsPerBlock はカーネルのレジスタ使用量によって変化するが，実行バイナリは nvcc のオプションでアーキテクチャごとに異なるものが生成されているため，同じカーネルであってもアーキテクチャによって変化する．4 つのカーネルの中で，GEMV-T は $TxStep = 32$ という制約があるため候補数が少ない．

4.3 実験結果

図 8 に SAXPY，**図 9** に SGEMV-N，**図 10** に SGEMV-T，**図 11** に STRMV-LNN の結果を示す．

はじめに図の読み方を説明する．まず上側に示す (a) の性能の図において，緑色の箱ひげ図（All）は取りうるすべてのスレッドブロックサイズに対する性能を示したものである．上から性能の最大値，第 3 四分位，中央値，第 1 四分位，最小値を表している．性能のばらつきは，スレッドブロックサイズの選択がカーネルの性能に与える影響の強さを示している．また最大値と他の値とのギャップは，最適なスレッドブロックサイズ選択の難しさを示唆している．次に赤線（Online）はオンライン自動チューニングによるスレッド数自動選択付き MUBLAS ルーチンの性能を示している．黄線（Offline）は自動チューニングによって選択されたスレッド数を前もってテーブルに格納し，そのスレッド数を静的に指定してカーネルを起動した場合の性能である．つまり，オンライン自動チューニングを用いていない「オフライン自動チューニング相当」の性能である．したがって，赤線と黄線の差はオンライン自動チューニングのコストを示している．青線（CUBLAS）は CUBLAS 7.0 の性能である．下側に示す (b) のスレッドブロックサイズの図では，実線はオンライン自動チューニングによって選択されたスレッドブロックサイズ，破線は最大性能となるスレッドブロックサイズ，すなわち (a) の箱ひげ図の最大値を記録したスレッドブロックサイズを示している．桃色が x 次元のスレッド数，青色が y 次元のスレッド数であり，SAXPY と SGEMV-T はスレッドブロックが 1 次元であるから，y 次元のスレッド数は常に 1 となっている．

まず SAXPY（図 8）であるが，性能曲線が山を描くのは L2 キャッシュの影響である．(a) の箱ひげ図から，最低性能だけが突出していることがわかる．これはスレッド数が $ThMin$ より小さい場合に性能が低下していることに起因する．AXPY はカーネルが非常に軽量であり，スレッドブロックの実行時間が短く Grid-Occupancy の影響を受けにくい．そのためスレッド数が $ThMin$ 以上であれば，スレッド数の選択が性能に与える影響は小さくなっている．また，問題サイズが小さい場合にはカーネル本体の実行時間が短くなるため，オンライン自動チューニングのコストが見えている．したがって，SAXPY の場合，オンライン自動チューニングを導入して問題サイズごとにスレッド数を調整する利点は薄いと言える．GeForce GTX 980 では選択されたスレッド数が性能の中央値を下回っているケースが見られる．例えば N=204800 において自動チューニングが選択したスレッド数と最適なスレッド数における 3 つの

表 3 オフライン自動チューニングにより得られた各 GPU における各ルーチンのチューニングレシピ（0 は最小値が存在しないことを示す）

GeForce GTX 580

	SAXPY	SGEMV-N	SGEMV-T	STRMV-LNN
$WrpOcpMin$	0.33	0.33	0.49	0.45
$BlkOcpMin$	0	0	0	0
$ThMin$	40	40	88	104
$TyPerTxMax$	–	1.9	–	2.0

Tesla K20c

	SAXPY	SGEMV-N	SGEMV-T	STRMV-LNN
$WrpOcpMin$	0.49	0.60	0.49	0.68
$BlkOcpMin$	0	0.12	0	0.24
$ThMin$	40	248	0	120
$TyPerTxMax$	–	0.5	–	6.0

GeForce GTX 980

	SAXPY	SGEMV-N	SGEMV-T	STRMV-LNN
$WrpOcpMin$	0	0	0	0.68
$BlkOcpMin$	0.09	0	0	0
$ThMin$	24	24	0	40
$TyPerTxMax$	–	16.0	–	10.5

表 4 スレッドブロックサイズの候補数（「想定上の最大」は起動できるスレッド数の制約およびチューニングレシピの $ThMin$ による制約を受けない場合の最大候補数）

	SAXPY	SGEMV-N	SGEMV-T	STRMV-LNN
想定上の最大	128	654	654	654
GeForce GTX 580	124	459	31	474
Tesla K20c	124	338	32	524
GeForce GTX 580	126	642	32	593

Occupancy を比較すると，Block-Occupancy のみ最適なスレッド数の場合より小さい 0.094 であった．$BlkOcpMin$ は満たしているが，チューニングレシピの生成方法に課題がある可能性がある．

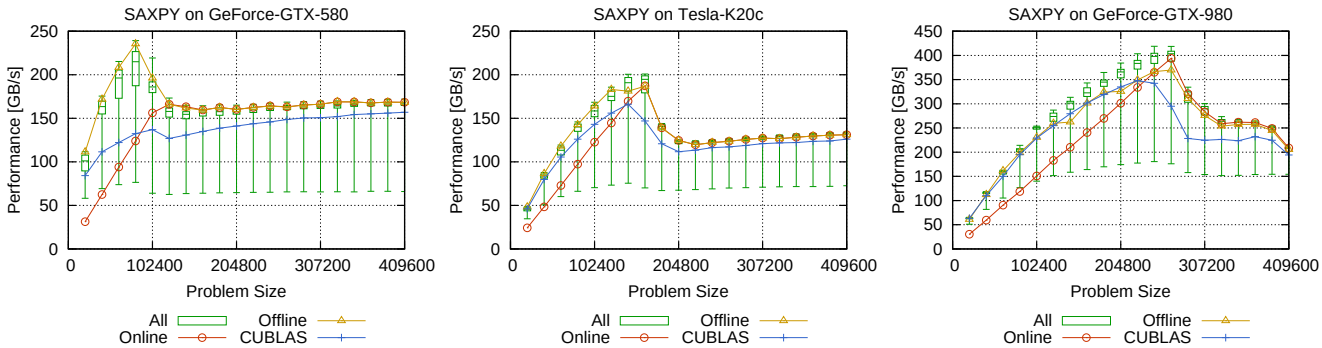
次に SGEMV-N（図 9）では，(a) の箱ひげ図から，スレッド数の選択が性能に与える影響が GPU の種類によって異なり，アーキテクチャが新しくなるほどその影響は小さくなっていることがわかる．また，問題サイズが小さいほどスレッド数の選択が難しい傾向にあることがわかる．これは問題サイズが小さい場合に，起動するスレッドブロック数が少なくなり，Grid-Occupancy を大きくするスレッドブロックサイズの候補が少なくなるからである．自動チューニングは多くの場合に最大値から第 3 四分位程度の性能を達成している．自動チューニングの有用性は問題サイズが小さい時ほど高いといえるが，問題サイズが小さいところではオンライン自動チューニングのコストが表れており，探索空間の枝刈りなどのコスト削減の工夫が求められる．一方，(b) のスレッドブロックサイズの図からは，自動選択されたスレッド数とベストなスレッド数は大きく異なっているにも関わらず，自動選択されたスレッド数によって最大性能に近い性能が実現されていることがわかる．

SGEMV-T（図 10）は SAXPY と同様にスレッド数選択が性能に与える影響は小さく，問題サイズに対してスレッド数を可変させる必要性は小さいと言える．また SGEMV-N と同様にアーキテクチャが新しくなるほどその影響は小さくなっていることがわかる．

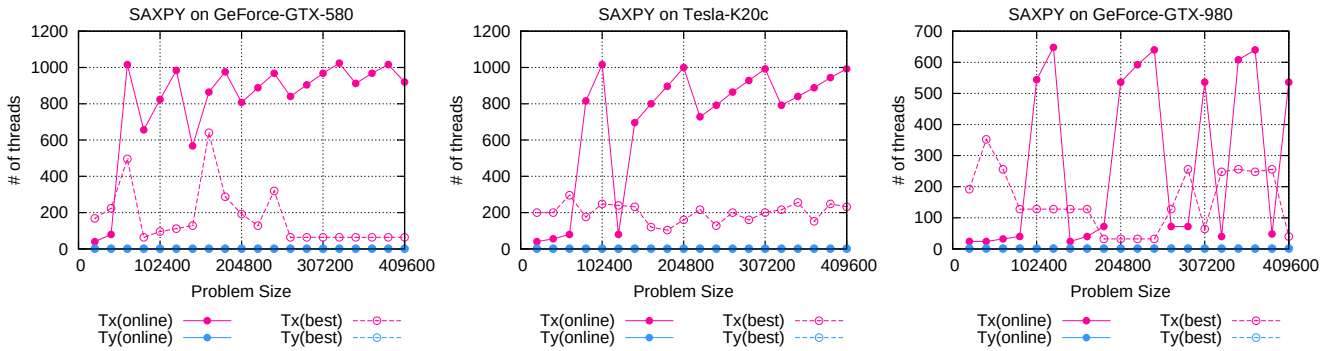
STRMV-LNN（図 11）は，SGEMV-N と似た実装であるが，三角行列を扱うために 4 つのカーネルの中ではもっとも複雑なメモリアクセスパターンを持つ．(a) の箱ひげ図より，SGEMV-N と同様にスレッド数の選択が性能に与える影響が大きいことがわかる．特に GeForce GTX 580 と Tesla K20c では最大値と第 3 四分位のギャップが他のルーチンと比べて大きく，最適なスレッド数の選択が難しいケースが多い．しかし自動チューニングによってカーネルが得られる最大性能に近い性能が達成できている．

5. まとめと今後の課題

本稿では著者らが開発している MUBLAS のメモリ律速な BLAS ルーチンにおいて，最適なスレッド数を自動チューニングで決定する方法を検討した．まず，CUDA カーネルの最適なスレッド数を決定するための理論を整理し，カーネルが最大効率で実行される状態を，ワープ，スレッドブロック，グリッドの 3 つのレベルにおける Occupancy として定義した．そしてこれら 3 つの Occupancy に基づい

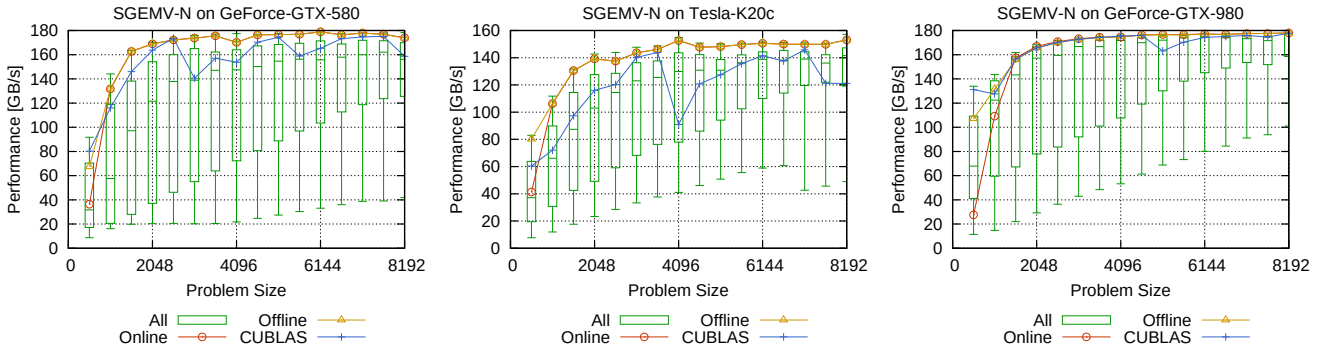


(a) 性能

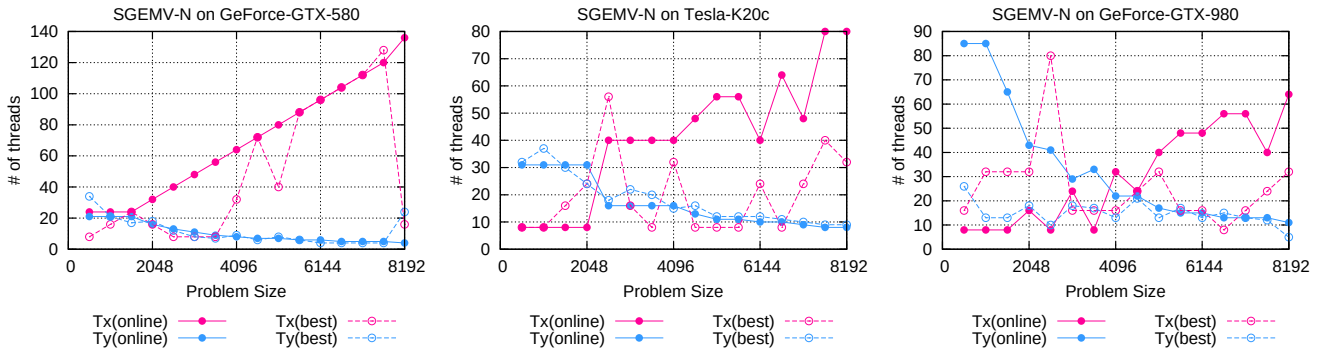


(b) スレッドブロックサイズ

図 8 SAXPY

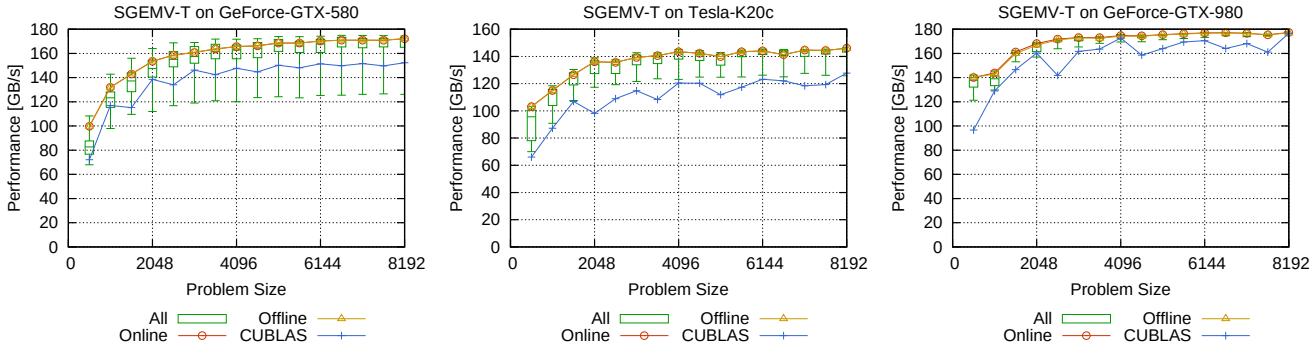


(a) 性能

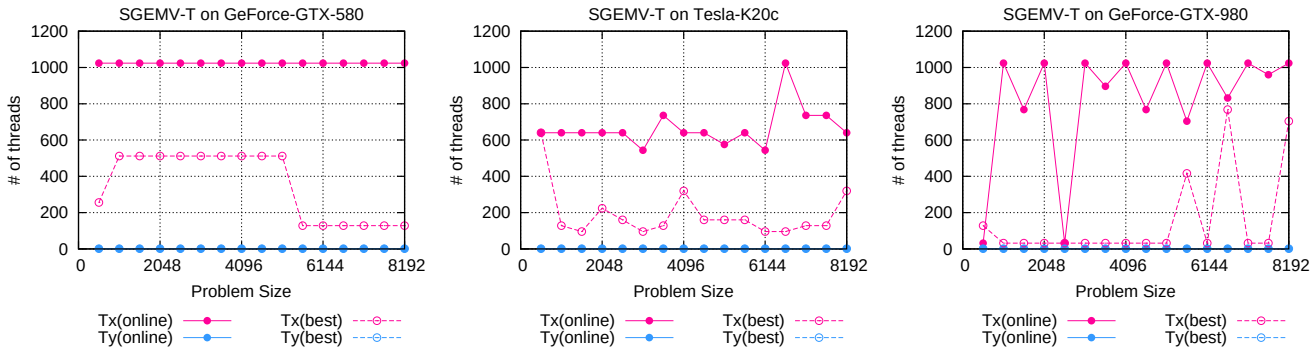


(b) スレッドブロックサイズ

図 9 SGEMV-N（スレッドブロックサイズ）

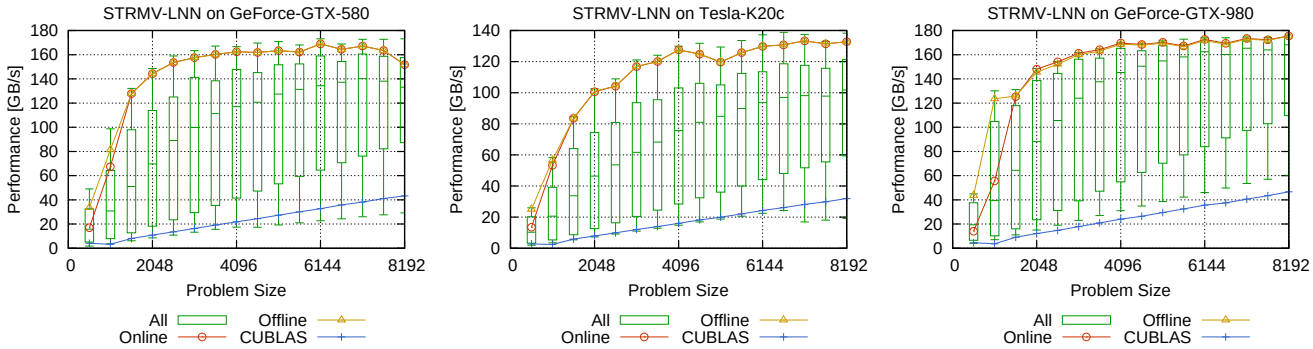


(a) 性能

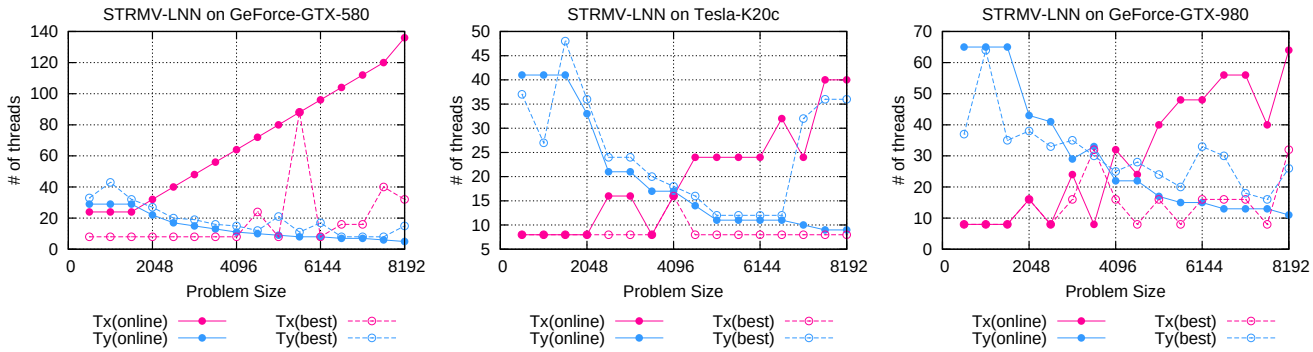


(b) スレッドブロックサイズ

図 10 SGEMV-T



(a) 性能



(b) スレッドブロックサイズ

図 11 STRMV-LNN

て最適なスレッド数を決定するために，2 つの自動チューニング機構を実装した．一つは BLAS のビルド時に実行されるオフライン自動チューニングであり，もう一つがルーチンが呼び出される度に実行されるオンライン自動チューニングである．前者のオフライン自動チューニングは，ワープとスレッドブロックに関する Occupancy の必要最小値およびスレッド数の取りうる範囲を，デバイスおよびカーネル固有のチューニングパラメータとして，カーネルのサンプリング実行から決定する．そして後者のオンライン自動チューニングでは，オフライン自動チューニングで決定したパラメータとグリッドレベルの Occupancy に基づいて，問題サイズに応じた最適なスレッド数を決定する．SAXPY，SGEMV-N，SGEMV-T，STRMV-LNN の各カーネルにおける評価では，Fermi，Kepler，Maxwell の 3 つのアーキテクチャにおいて，多くのケースでほぼ最適なスレッド数を選択できることを示した．

現状での課題を以下に示す．

- チューニングレシピの生成方法の妥当性：現状ではカーネルのサンプリング実行によって取得しているが，サンプリングの方法そのものについては，問題サイズの選び方や目標性能の与え方などに検討の余地がある．またサンプリング実行にはそれなりの時間を要する（例えば SGEMV-N では 1 時間以上）ため，コスト削減も課題である．そのため今後はマイクロベンチマークや理論ベースのモデル化を検討する．
- オンライン自動チューニングのコスト：カーネル実行時間が小さい場合にオンライン自動チューニングのコストが問題となっている．問題サイズが小さい場合にはスレッド数の最大値に対する制約を理論的に与えられる可能性がある．またカーネルのサンプリング実行から，オンライン自動チューニングを使わないという選択肢を取り入れることも検討できる．
- 異なるデバイス間における演算結果の bit レベルの一貫性の保証：カーネル実装によっては，デバイスによって選択されるスレッド数が異なると，総和演算などでスレッド数の選択が計算順序に影響を与える場合には，丸め誤差の影響により，異なるデバイス間で丸め誤差レベルで演算結果が異なる可能性がある．演算順序がスレッド数に依存しない実装の検討が必要である．また根本的な解決策ではないが，CUBLAS の SYMV ルーチンのように，演算結果の一貫性を保証するルーチンと，保証しない代わりに高速なルーチンの 2 種類を用意することで，ユーザに検証用の道具を提供する方法もある．

今後はこれらの課題の解決とともに，性能が演算律速となるようなカーネルを含むその他のカーネルへの適用を検討している．

謝辞 本研究は科研費（課題番号：15H02709）の助成を受けたものである．

参考文献

[1] RIKEN: MUBLAS 1.4.28, http://www.aics.riken.jp/labs/lpnctrt/ASPENK2.html.
[2] Mukunoki, D., Imamura, T. and Takahashi, D.: Fast Implementation of General Matrix-Vector Multiplication (GEMV) on Kepler GPUs, *Proc. 23rd Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP 2015)*, pp. 642–650 (2015).
[3] 須田礼仁：オフライン自動チューニングの数理手法，情報処理学会研究報告，Vol. 2010-HPC-125, No. 3, pp. 1–9 (2010).
[4] NVIDIA Corporation: The NVIDIA CUDA Basic Linear Algebra Subroutines, https://developer.nvidia.com/cublas.
[5] Innovative Computing Laboratory, University of Tennessee: Matrix Algebra on GPU and Multicore Architectures, http://icl.cs.utk.edu/magma.
[6] Abdelfattah, A., Keyes, D. and Ltaief, H.: KBLAS: An Optimized Library for Dense Matrix-Vector and Matrix-Matrix Operations on GPU Accelerators, http://ecrc.kaust.edu.sa/Pages/Res-kblas.aspx.
[7] Abdelfattah, A., Keyes, D. and Ltaief, H.: Systematic Approach in Optimizing Numerical Memory-Bound Kernels on GPU, *Euro-Par 2012: Parallel Processing Workshops*, Lecture Notes in Computer Science, Vol. 7640, pp. 207–216 (2013).
[8] 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（ACS），Vol. 4, No. 4, pp. 158–168 (2012).
[9] Xu, W., Liu, Z., Wu, J., Ye, X., Jiao, S., Wang, D., Song, F. and Fan, D.: Auto-Tuning GEMV on Many-Core GPU, *Proc. 2012 IEEE 18th International Conference on Parallel and Distributed Systems (ICPADS '12)*, pp. 30–36 (2012).
[10] Sørensen, H. H. B.: Auto-tuning Dense Vector and Matrix-Vector Operations for Fermi GPUs, *Parallel Processing and Applied Mathematics (PPAM 2011)*, Lecture Notes in Computer Science, Vol. 7203, pp. 619–629 (2012).
[11] Kurzak, J., Tomov, S. and Dongarra, J.: Autotuning GEMM Kernels for the Fermi GPU, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 23, No. 11, pp. 2045–2057 (2012).
[12] RIKEN: ASPEN.K2, http://www.aics.riken.jp/labs/lpnctrt/ASPENK2.html.
[13] NVIDIA Corporation: CUDA C PROGRAMMING GUIDE, PG-02829-001_v7.0, http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf (2015).
[14] NVIDIA Corporation: CUDA C BEST PRACTICES GUIDE, DG-05603-001_v7.0, http://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf (2015).