Level-3 BLAS に基づく固有値解法のマルチコアプロセッサ・GPU上での性能

2012年 5月 28日

研究会「大規模シミュレーションと数理アルゴリズム

神戸大学大学院システム情報学研究科 計算科学専攻
／JST-CREST

山本　有作

はじめに：本研究で扱う問題

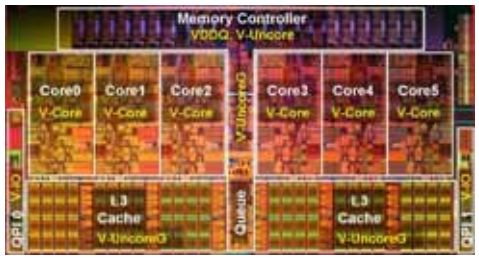
- **実対称密行列の固有値分解** $A = Q\Lambda Q^{\mathrm{T}}$
  - A：$n \times n$ **実対称密行列**, Λ：**対角行列**, Q：**直交行列**
  - **応用分野**
    - **量子化学**
    - **統計計算**

- **非対称密行列の固有値分解** $A = X\Lambda X^{-1}$
  - A：$n \times n$ **非対称密行列**, Λ：**対角行列**, X：**正則行列**
  - **応用分野**
    - **電子線回折**
    - **構造計算**
      - 超大規模疎行列を中規模密行列に縮約してモード解析

本研究の目的

- マルチコア／メニーコアプロセッサ上で高性能を達成できる密行列向け固有値ソルバの開発

- 背景
  - 問題の大規模化
  - マルチコア／メニーコア／GPGPUの普及
  - ポストペタスケール計算機はメニーコアベース？



Intel Core i7-980X

ClearSpeed CSX600

NVIDIA Tesla

目次

実対称固有値問題の
マルチコア向け最適化

実対称密行列に対する標準的な固有値計算アルゴリズム

実対称密行列 A

	計算内容	計算手法
3重対角化	$Q^T A Q = T$ (Q: **直交行列**)	**ハウスホルダー法**
3**重対角行列** T		
3重対角行列の 固有値・固有ﾍﾞｸﾄﾙ計算	$T\,\boldsymbol{u}_i = \lambda_i \boldsymbol{u}_i$	QR**法** **分割統治法** MR^3**アルゴリズム**
T **の固有値** $\{\lambda_i\}$**, 固有ベクトル** $\{\boldsymbol{u}_i\}$		
逆変換	$\boldsymbol{v}_i = Q\,\boldsymbol{u}_i$	**逆変換**

A**の固有ベクトル** $\{\boldsymbol{v}_i\}$

各部分の演算量と実行時間

密行列 A

↓

3重対角化

↓

3重対角行列の
固有値・固有ﾍﾞｸﾄﾙ計算

↓

逆変換

↓

Aの固有ベクトル $\{v_i\}$

演算量

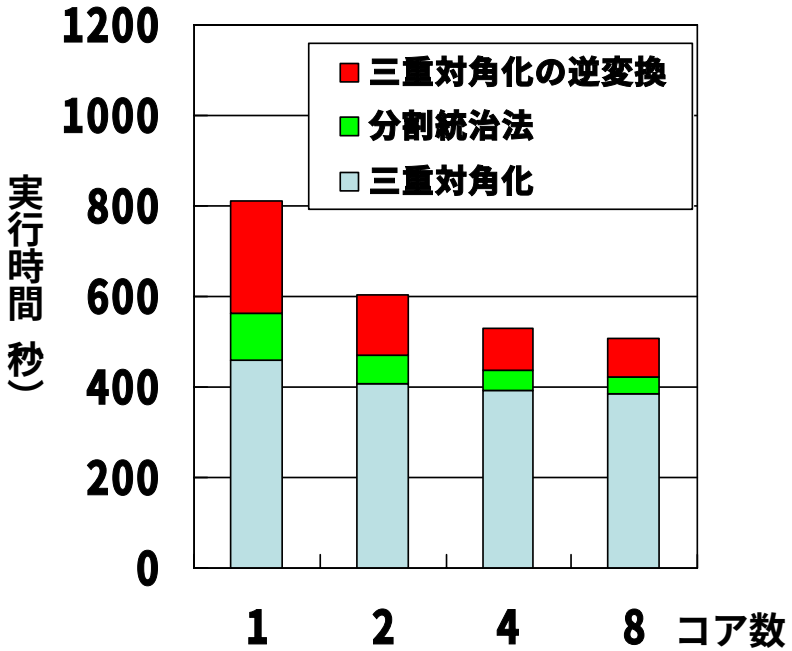
$(4/3)\, n^3$

$O(n^2)$
〜 $O(n^3)$

$2mn^2$
（m本の固有
ベクトル）

実行時間(全固有ﾍﾞｸﾄﾙ)

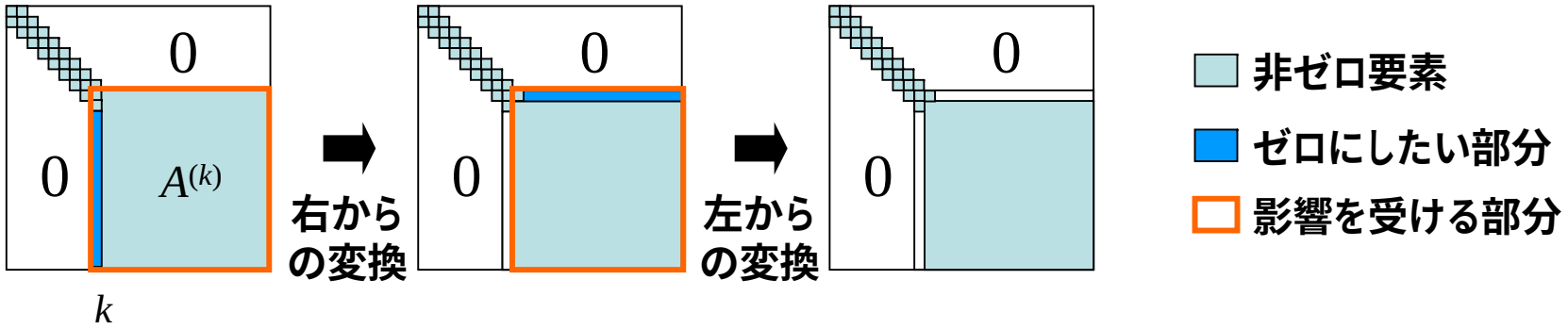
$n = 9000$，Quad Core Xeon ×2
LAPACK での実行時間（秒）



- 3重対角化が実行時間の大部分を占める
- 速度向上率が低い

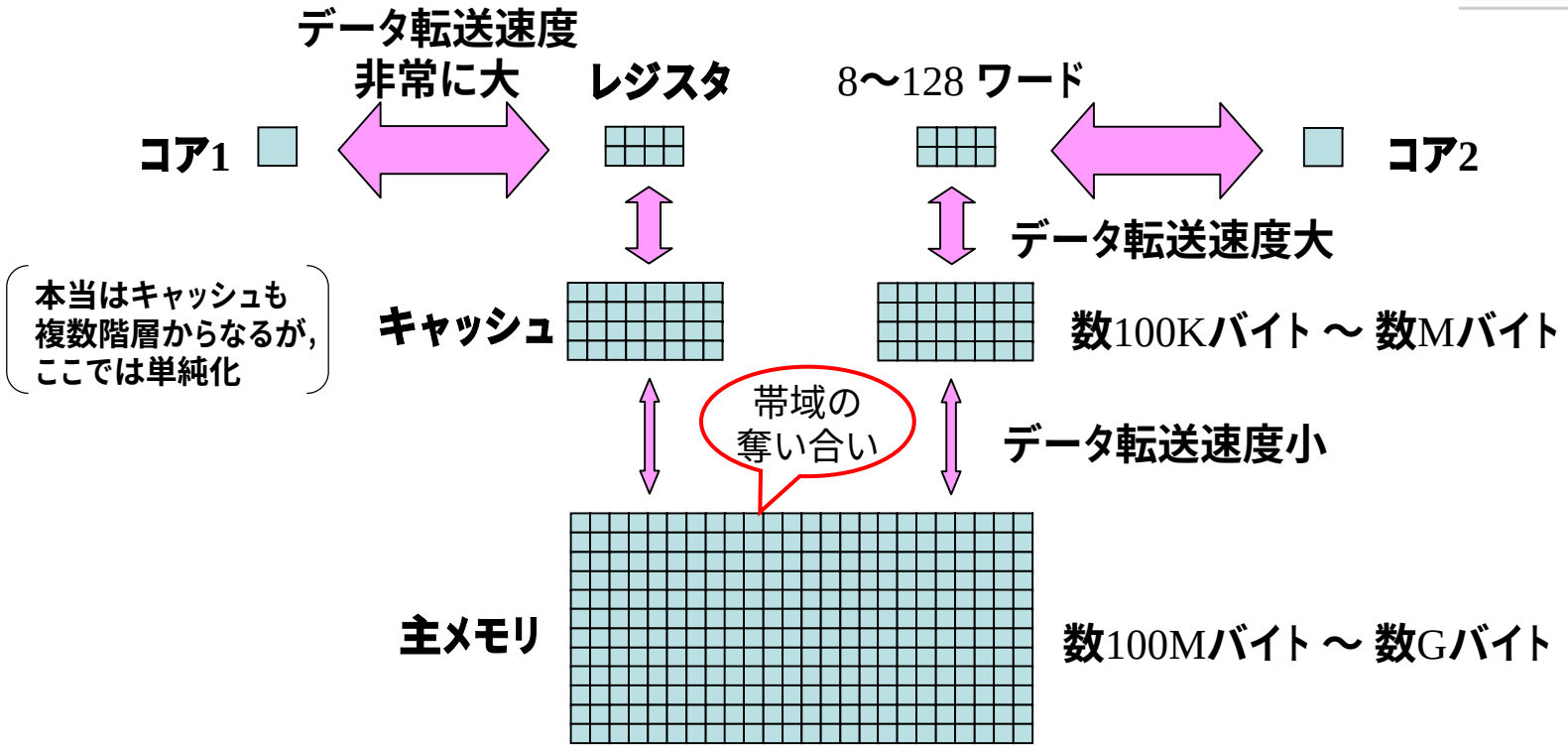
3重対角化の性能が出ない原因

- **3重対角化の演算パターン**
  - **左右からのハウスホルダー変換による行･列の消去**
    - $A^{(k)} := (I - \alpha \boldsymbol{w}\boldsymbol{w}^T)\, A^{(k)} = A^{(k)} - \alpha \boldsymbol{w}\, (\boldsymbol{w}^T A^{(k)})$ 行列ベクトル積
      Rank-1更新
    - **演算は** level-2 BLAS **(行列ベクトル積と** rank-1 **更新)**



- **演算パターンに関する問題点**
  - Level-2 BLAS **はデータ再利用性が低い。**
    - ⇨ **キャッシュの有効利用が困難であり，単体性能が出にくい。**
    - ⇨ **プロセッサ間のアクセス競合により，並列性能向上も困難**

マルチコアプロセッサのメモリ階層

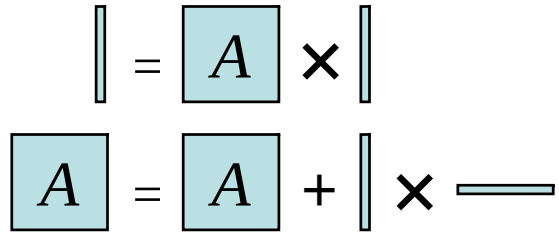


- 主メモリアクセスの場合，実効性能は実質的に転送速度で決まる
  - Byte/Flop値: 1演算を行う時間で転送できるバイト数 $\lesssim 1$
- マルチコアでは複数のコアが帯域を奪い合うため，問題が更に深刻化
- 主メモリのデータ転送速度の遅さをカバーするには，キャッシュにデータがある間に演算をまとめて行う（データの再利用）ことが必要

BLASの利用による高性能化

- BLASとは
  - Basic Linear Algebra Subprograms の略
  - 個々のマシン向けにチューニングした基本行列演算のライブラリ

- BLASの種類
  - Level-1 BLAS: ベクトルとベクトルの演算
    - 内積 $c := x^T y$
    - AXPY演算 $y := ax + y$ など
  - Level-2 BLAS: 行列とベクトルの演算
    - 行列ベクトル積 $y := Ax$
    - 行列のrank-1更新 $A := A + xy^T$

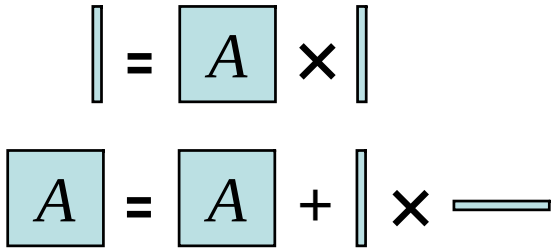


  - Level-3 BLAS: 行列と行列の演算
    - 行列乗算 $C := AB$ など

BLASにおけるデータ再利用性と並列粒度

- Level-1 BLAS
  - 演算量： $O(N)$， データ量： $O(N)$
  - データ再利用性： なし
  - 並列粒度： $O(N/p)$ （N：ベクトルの次元，p：プロセッサ台数）

- Level-2 BLAS
  - 演算量： $O(N^2)$， データ量： $O(N^2)$
  - ベクトルデータのみ再利用性あり
  - 並列粒度： $O(N^2/p)$

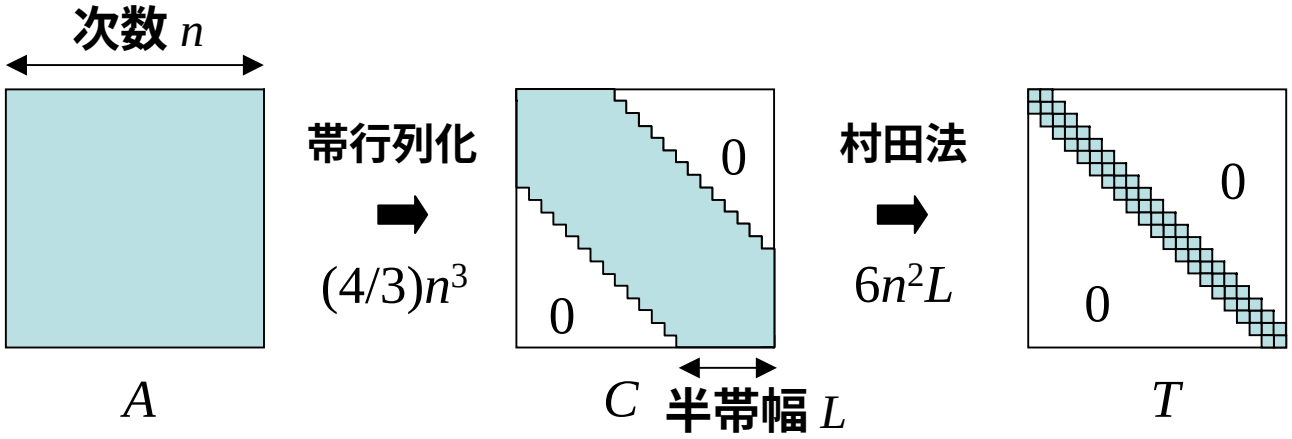


- Level-3 BLAS
  - 演算量： $O(N^3)$， データ量： $O(N^2)$
  - $O(N)$回のデータ再利用が可能
    - 行列乗算のサイズを大きくするほど，再利用性が向上
    - Byte/Flop 値の小さいマシンでも性能を出せる。
  - 並列粒度： $O(N^3/p)$

演算をできる限り level-3 BLAS で行うことが高性能化のポイント

Level-3 BLAS に基づく固有値分解アルゴリズム

- **2段階の3重対角化アルゴリズム** (Bischof et al., '93)
  - **密行列 A をまず帯幅 L の帯行列 C に変換**
  - **次にこの帯行列を3重対角行列 T に変換**



- **3重対角化を2段階で行うことの利点**
  - **前半の変換は, level-3 BLAS (行列乗算) のみを使って実行可能**
    - ⇨ **キャッシュの有効利用が可能**
  - **後半の変換は level-2 BLAS が中心だが, 演算量は $O(n^2L)$**
    - **前半部に比べてずっと小さい。**

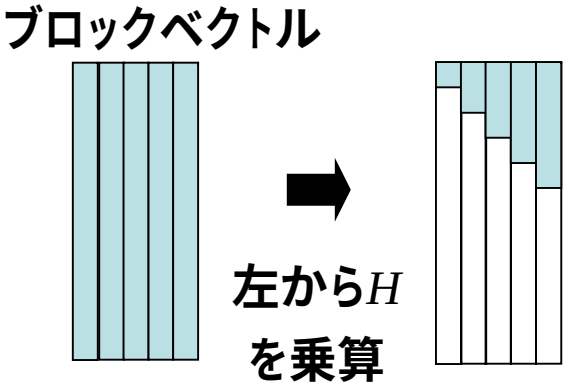
帯行列化のアルゴリズム

ブロック鏡像変換によるブロック列の消去

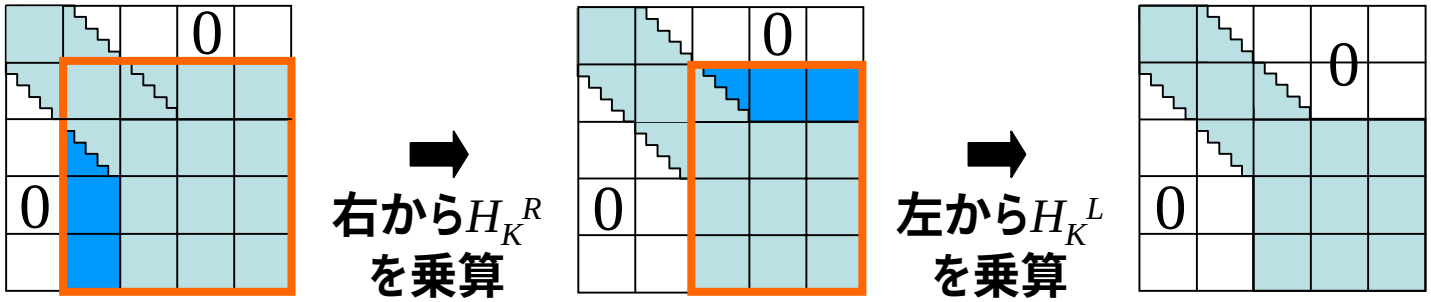
– ブロック鏡像変換 $H = I - W\alpha W^T$

$W\alpha W^T$: 行列

- H は直交行列
- 与えられたブロックベクトルを上三角行列（正確には右上三角部分のみ非零でそれ以外が零の行列）に変形



第 K ステップでの処理

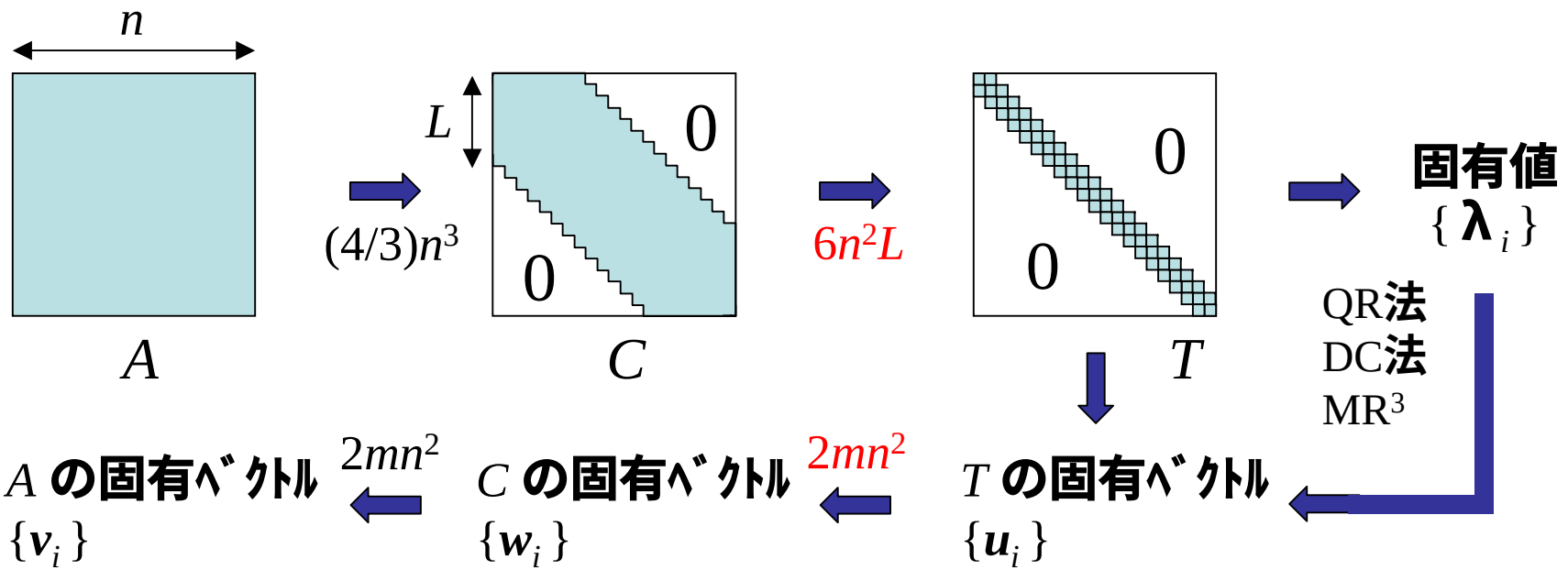


非ゼロ要素　ゼロにしたい部分　影響を受ける部分

- 消去演算は**行列乗算のみ**で行える
- 行列乗算のサイズ ～ L

Level-3 BLAS に基づく固有値計算アルゴリズム

- **固有ベクトル計算を含めたアルゴリズムの全体像**



- **長所**
  - $O(n^3)$ の演算量を持つ部分はすべて level-3 BLAS で実行可能
- **短所**
  - 逆変換の演算量が $4mn^2$ (従来法の2倍)。ただし level-3 化可能

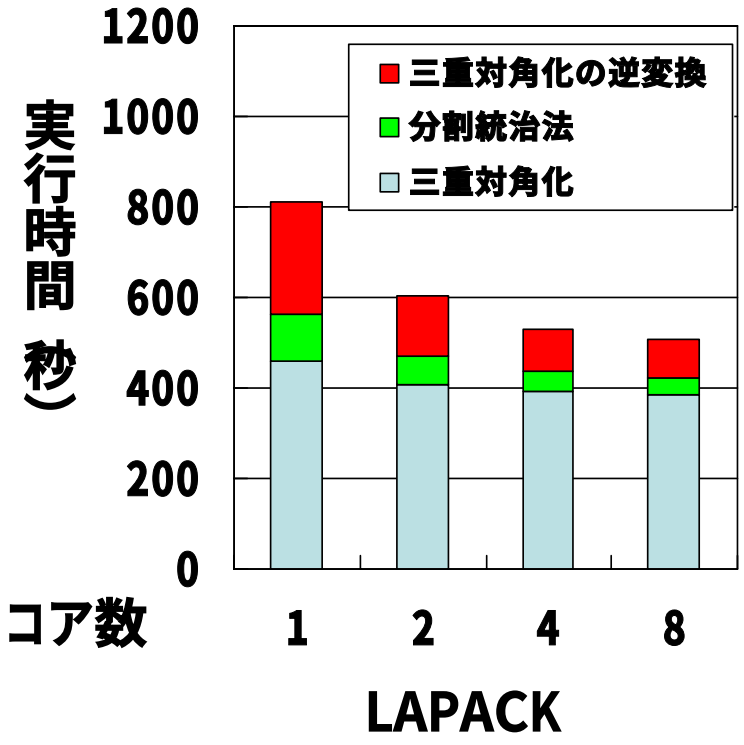
性能評価

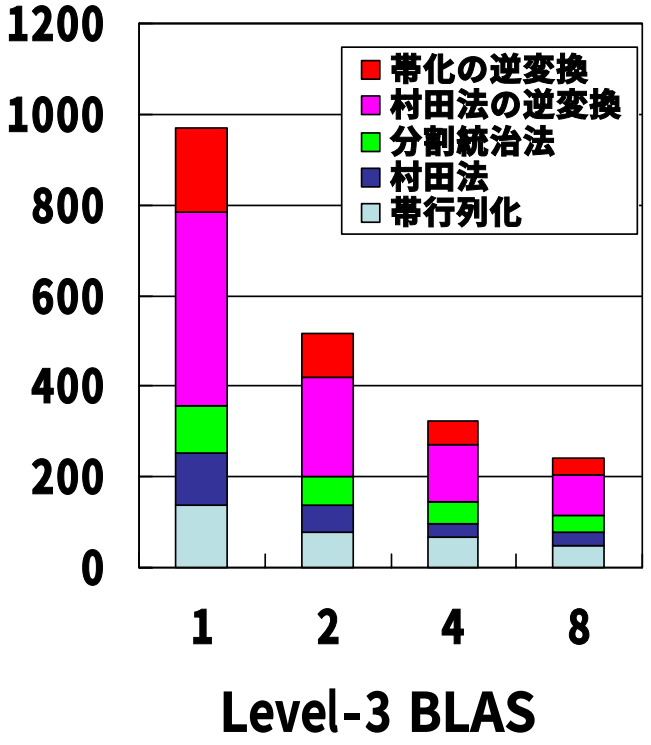
- **評価条件**
  - **問題サイズ** n : 9000，　**半帯幅** L : **環境に応じて最適化**
  - **全固有値･固有ベクトルを計算**
  - Level-3 BLAS **に基づく手法は** Fortran **で作成**
  - **従来法は**LAPACK**のルーチンを使用**
- **計算機環境**
  - Xeon 8**コアマシン**
    - Xeon X5355 (2.66GHz, Quad-core) **×2ソケット**
    - Intel Fortran Compiler 9, Intel Math Kernel Library
  - **富士通** HX600　1**ノード (名大情報連携基盤センター)**
    - Opteron (2.5GHz, Quad-core) **×4ソケット**
    - **富士通 Fortran Compiler, SSL II**
  - Opteron 24**コアマシン**
    - Opteron 8431 (2.4 GHz, Hexa-core)**×4 ソケット**
    - GNU Fortran Compiler 4.4.0, GotoBLAS2 1.13

実験協力: **廣田悠輔 (神戸大学博士課程3年)**

Xeon 8コアマシンでの性能

- $n = 9000$, 半帯幅 $L = 100$



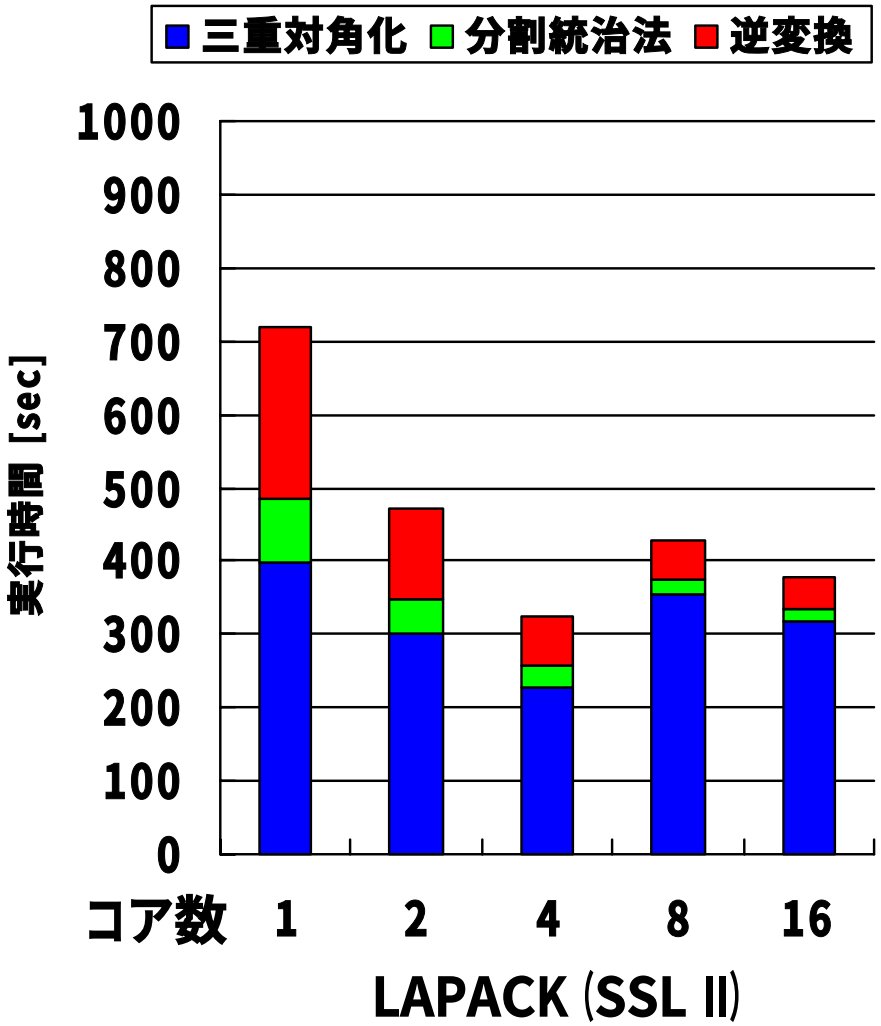


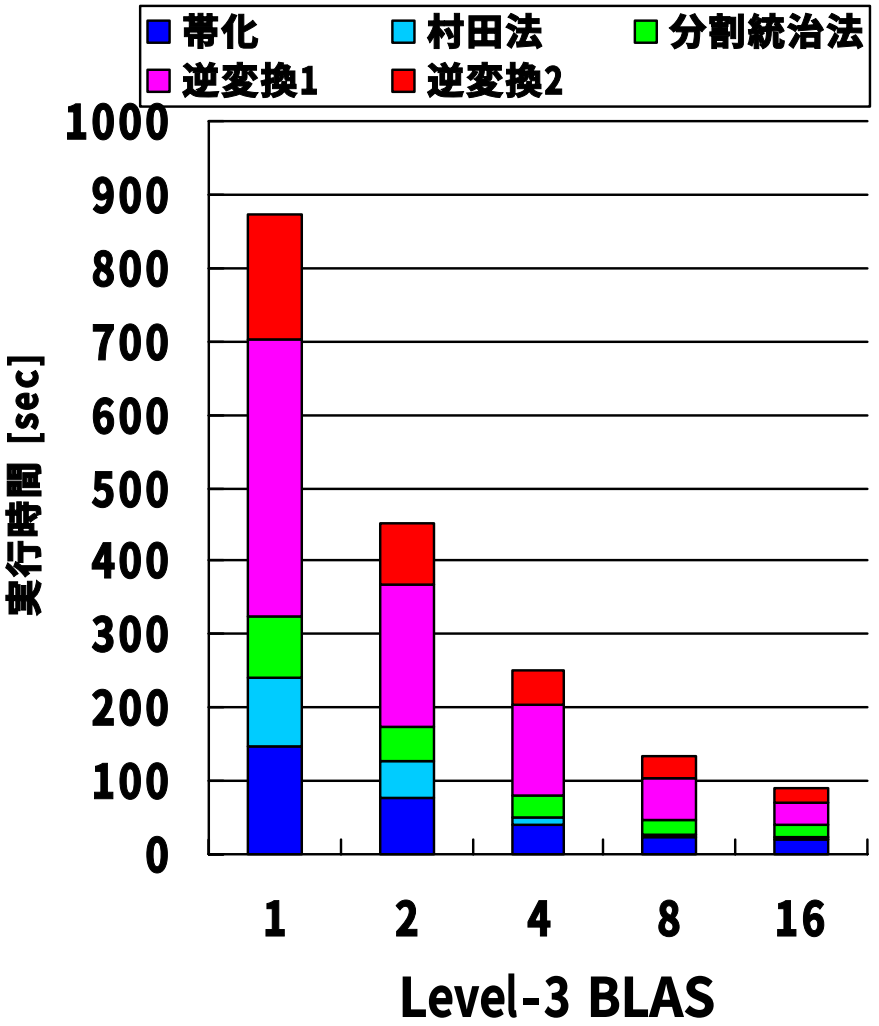
- **結果**
  - 8コアの場合, Level-3 はLAPACK と比べて最大2.1倍高速
  - Level-3 では, 実行時間の半分以上を逆変換が占める

⇨ 必要な固有ベクトルの本数が少ない場合は, さらに有利

富士通 HX600 での性能

- $n = 9000$, 半帯幅 $L = 50$

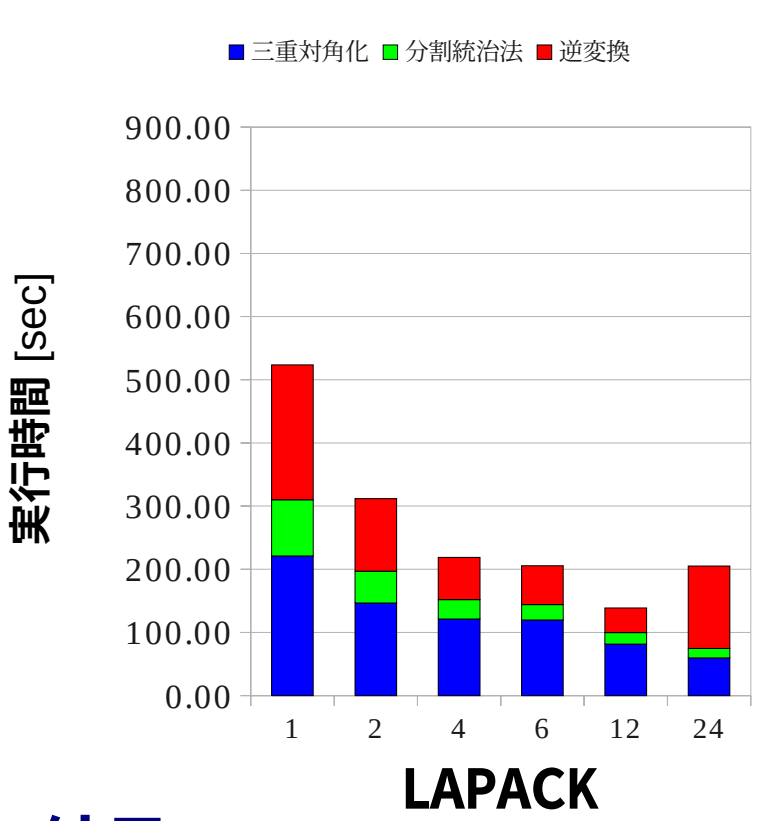


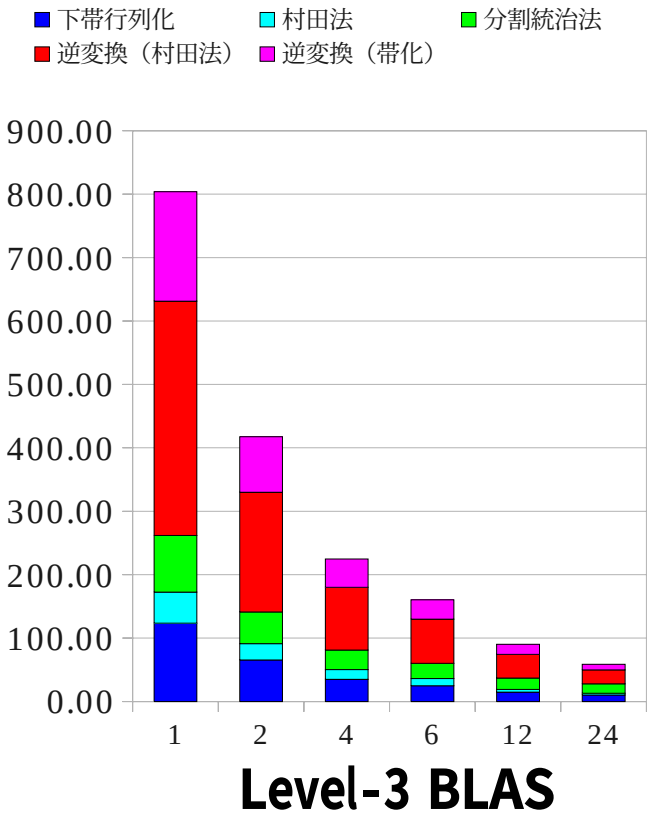


Level-3 はLAPACK と比べて最大 3.6 倍高速

Opteron 24コアマシンでの性能

- $n = 9000$, **半帯幅** $L = 100$, $MB = MB_2 = 1$





- **結果**
  - **Level-3 BLAS版（24コア）はLAPACK（12コア）と比べて2.4倍高速**
  - **Level-3 BLAS版（24コア）は，実行時間の約50%を2段階の逆変換が占める**

実行環境の詳細

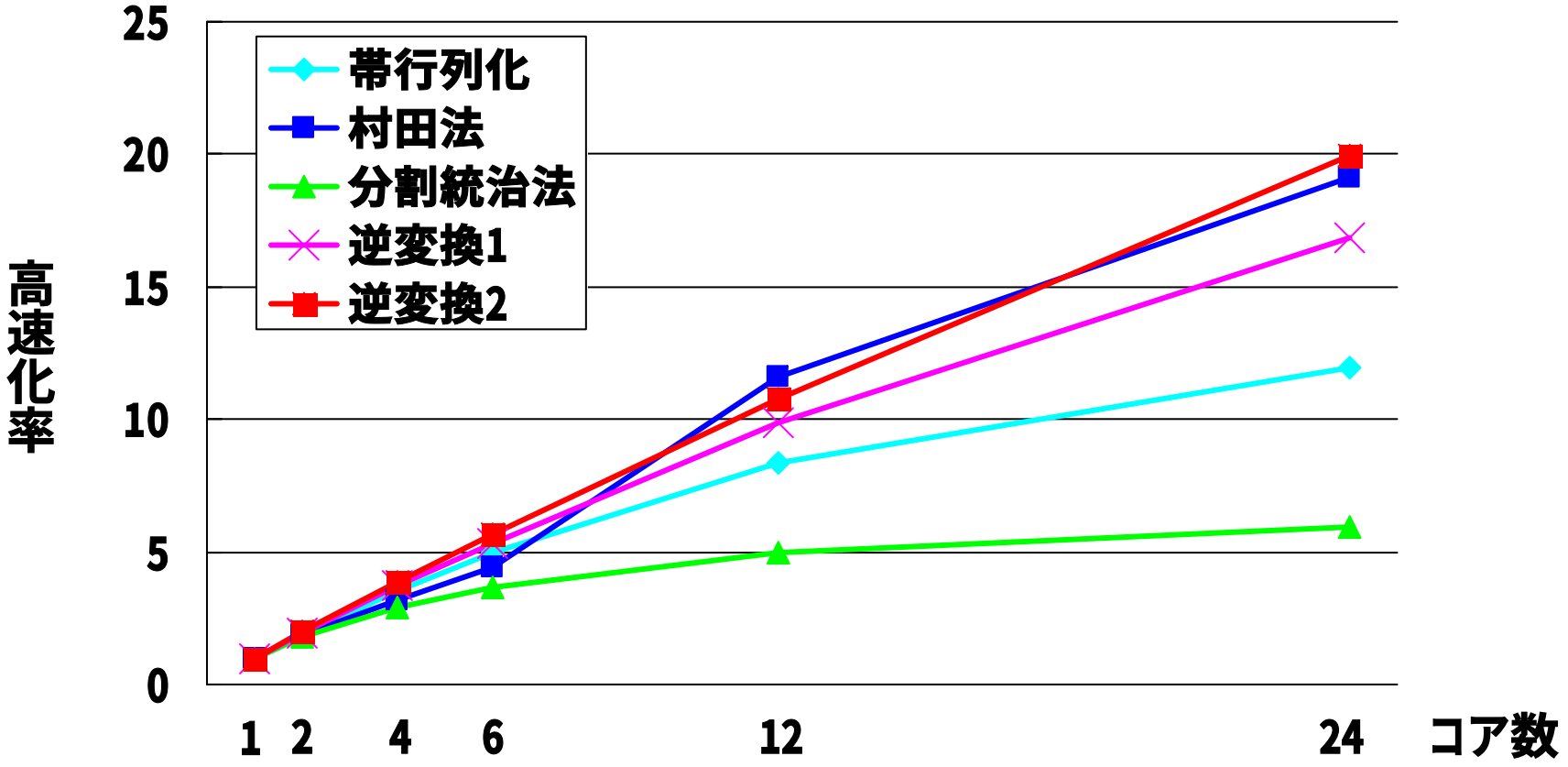
- **ハードウェア**
  - Opteron 8431 (2.4 GHz, 6**コア**)×4 **ソケット**
  - **コアあたりピーク性能** 9.6 GFLOPS
  - L1$: 128KB/**コア**, L2$: 512KB/**コア**, L3$: 6MB/**ソケット**
  - **主記憶容量**: 8 GB×4

- **ソフトウェア**
  - OS: CentOS 5.5 (x86_64)
  - **コンパイラ**: GNU Fortran Compiler 4.4.0
  - LAPACK/BLAS: GotoBLAS2 1.13

- numactl
  - 1,2,4,6**コア**： numactl --cpunodebind=2 –membind=2
  - 12**コア**： numactl --cpunodebind=0,1 --interleave=0-1
  - 24**コア**： numactl --interleave=all

Opteron 24コアマシンでの高速化率

- $n = 9000$, 半帯幅 $L = 100$



- 村田法, 逆変換1, 2は24コアまでほぼ順調に加速
- 帯行列化の加速率はやや低い
- 分割統治法 (LAPACK) の加速は不十分

⇒ 改良要

ライブラリ化に向けて

- **星先生との議論**
  - 大規模電子状態計算ソフトウェア ELSES への組み込みに向けた検討を実施中
  - ソルバのインターフェース仕様の案を策定
    - LAPACK 互換インターフェース
    - 独自インターフェース
  - ソルバの使い方がわかるようなサンプルプログラムを用意
- **スケジュール**
  - 今年度中にマルチコア版ソルバを公開の予定
    - JST-CRESTプロジェクト「ポストペタスケールに対応した階層モデルによる超並列固有値解析エンジンの開発」の支援による
- **自動チューニング機能の付加**
  - 問題や計算機環境に応じて最適な解法･パラメータを自動的に学習する機能を，今後開発予定
  - 同じサイズの固有値問題を多数回解く場合に有効

まとめ

- 密行列に対する実対称固有値問題の標準的なアルゴリズムでは，3重対角化の部分で level-2 BLAS を多用するため，マルチコアプロセッサで十分な性能を得られない
- Level-3 BLAS 版のアルゴリズムでは，演算量は増加するが，キャッシュの利用効率向上，アクセス競合の回避により，従来法より高い性能が得られる
- 必要な固有値･固有ベクトルが少ない場合，level-3 BLAS 版のアルゴリズムはさらに有利となる
- Level-3 BLAS 版のアルゴリズムのうち，帯行列化と逆変換の部分については，加速率向上のために更なる改良が必要である
- 今年度中にマルチコア版ソルバをライブラリ化し，公開の予定

今後の課題

- 帯行列化，分割統治法部分の性能分析と改良
- GPU／メニーコアプロセッサ向けの実装
- マルチコア／メニーコア／GPUをノードとする分散メモリ型並列計算機向けの実装
  - 電気通信大学　今村俊幸先生と協力
- 解法の性能モデリングとパラメータの自動チューニング方式の開発

非対称固有値問題の GPU向け最適化

共同研究者：村松淳一（名古屋大学修士修了）

非対称密行列に対する固有値計算アルゴリズム

- **ヘッセンベルグQR法**

Step 1 : **ヘッセンベルグ行列への相似変換**

$$Q_{n-2}\cdots Q_1 \boxed{A} Q_1^T\cdots Q_{n-2}^T = \boxed{H}$$

密行列　　ヘッセンベルグ行列

Step 2 : **直交変換の蓄積**

$$Q \leftarrow Q_{n-2}\cdots Q_1$$

Step 3 : QR**法による固有値の計算**

$$P_l\cdots P_1 \boxed{H} P_1^T\cdots P_l^T \approx \boxed{T}$$

ヘッセンベルグ行列　　上三角行列
(対角要素が固有値)

$$Q \leftarrow P_l\cdots P_1 Q$$

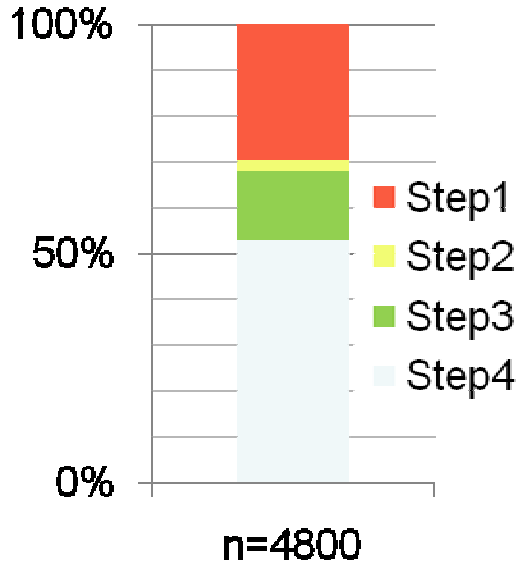
Step 4 : **固有ベクトルの計算**

$T\boldsymbol{y}_i = \lambda_i \boldsymbol{y}_i$ ： 逆反復法による T の固有ベクトルの計算

$\boldsymbol{x}_i \leftarrow Q^T \boldsymbol{y}_i$ ： 逆変換により A の固有ベクトルを求める

非対称密行列に対する固有値計算アルゴリズム (続き)

- **計算時間の内訳**
  - Step 1 と 4 が多くを占める
  - Step 1
    - Level-2 および 3 の BLAS で構成
      - 行列ベクトル積
      - 行列乗算
    - Level-2 部分はメモリ転送速度により性能が決まる
  - Step 4
    - Level-1 BLAS 中心
    - 多数の条件分岐を含む複雑なコード



計算機環境:
CPU : Core i7 920 (2.66 GHz)
Memory: 6.0GB

GPU による汎用計算

- GPU (Graphics Processing Unit)
  - グラフィックス処理を主目的とした演算装置

NVIDIA Tesla C1060
・メモリ転送速度：
　102GB/s
・演算性能（理論値）
　倍精度：78GFLOPS
　単精度：933GFLOPS

- GPU の特徴
  - 高いメモリ転送性能
  - 高い演算性能
  - 単純な計算を大量に行うのが得意
  - 複雑な条件分岐は不得手

- GPU による汎用計算
  - 開発環境の充実：NVIDIA 社 CUDA
  - CUBLAS, CUFFT などのライブラリ群も整備

本研究では，GPU を利用して Step 1 のヘッセンベルグ化を加速

CUDAによる行列計算の高速化

◆ GPU向けにプログラムの移植

- 拡張されたC言語でコーディングして，nvccでコンパイル
- 自由度の高いプログラミングが可能
- GPUの性能を十分に引き出すには様々なチューニングが必要
  （スレッド数，条件分岐，メモリアクセスの連続性など）

◆ CUBLASの利用

- CUDAで提供されているBLAS（Basic Linear Algebra Subprograms）
- 標準のC言語のプログラム中で利用可能　（gccでコンパイル可能）
- 限られた基本演算（行列ベクトル積，行列乗算など）のみ
- GPU向けにチューニング済み

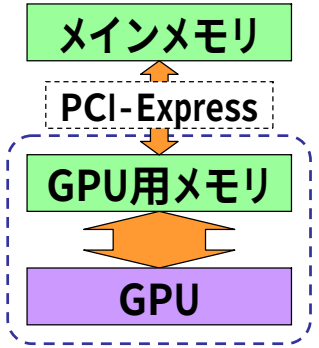
今回はCUBLASを使って高速化を行う

CUBLASの特徴

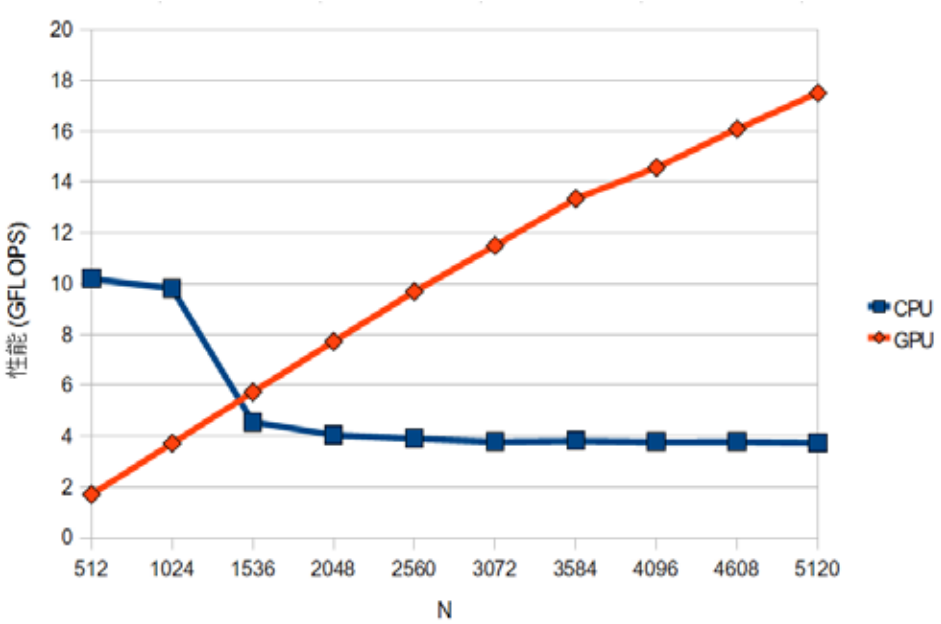
◆ CUBLAS の仕様

- ユーザー自身がデータ転送を制御
- ボードのデータはプログラム終了まで保持

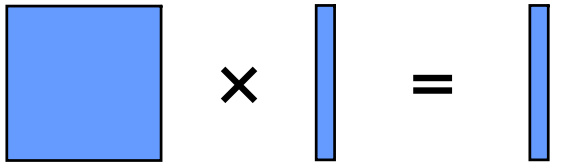
メモリ配置の工夫によるデータ転送コストの削減



◆ 性能



SGEMV (行列ベクトル積, level-2 BLAS)



小さいサイズでは CPU より低速
大きいサイズでは4倍以上高速

⇨ 大きいサイズの演算を GPU に担当させる

ヘッセンベルグ化の詳細

- **ハウスホルダー変換**

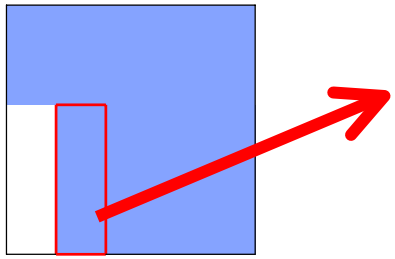
$$\underbrace{(I - t\boldsymbol{v}\boldsymbol{v}^T)}_{=H}\boldsymbol{a} = \boldsymbol{b}$$

- $\|\boldsymbol{a}\|_2 = \|\boldsymbol{b}\|_2$
- $\boldsymbol{v} = \boldsymbol{a} - \boldsymbol{b}$
- $t = \dfrac{2}{\|\boldsymbol{v}\|_2}$

$(H^T = H,\ H^T H = HH^T = I)$

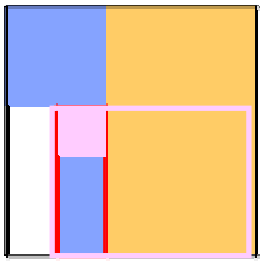
- **ハウスホルダー変換を用いたヘッセンベルグ化**

for $i = 1, N - 2$



$t_i, \boldsymbol{v}_i$ の計算

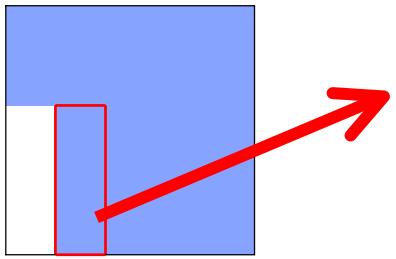
$(I - t_i \boldsymbol{v}_i \boldsymbol{v}_i^T)$



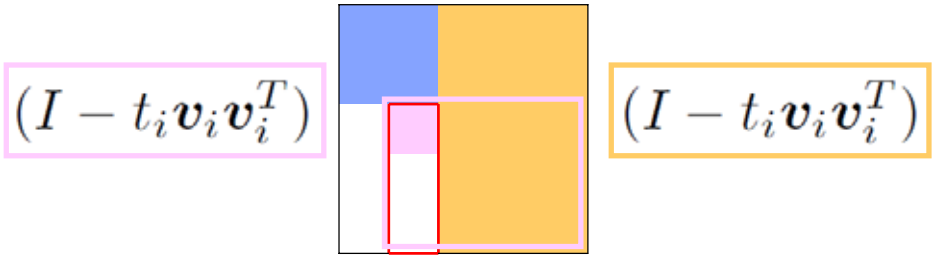
$(I - t_i \boldsymbol{v}_i \boldsymbol{v}_i^T)$

ヘッセンベルグ化の計算手順

for $i = 1, N - 2$



$t_i, \boldsymbol{v}_i$ の計算



$$(I - t_i \boldsymbol{v}_i \boldsymbol{v}_i^T)A \begin{cases} 1. & \boldsymbol{w}_i^T = t_i \boldsymbol{v}_i^T A \quad \text{(行列ベクトル積)} \\ 2. & A \leftarrow A - \boldsymbol{v}_i \boldsymbol{w}_i^T \quad \text{(行列のRank-1更新)} \end{cases}$$

これらの演算は、データアクセス回数と演算量が同じオーダー

$O(N^2)$　　$O(N^2)$

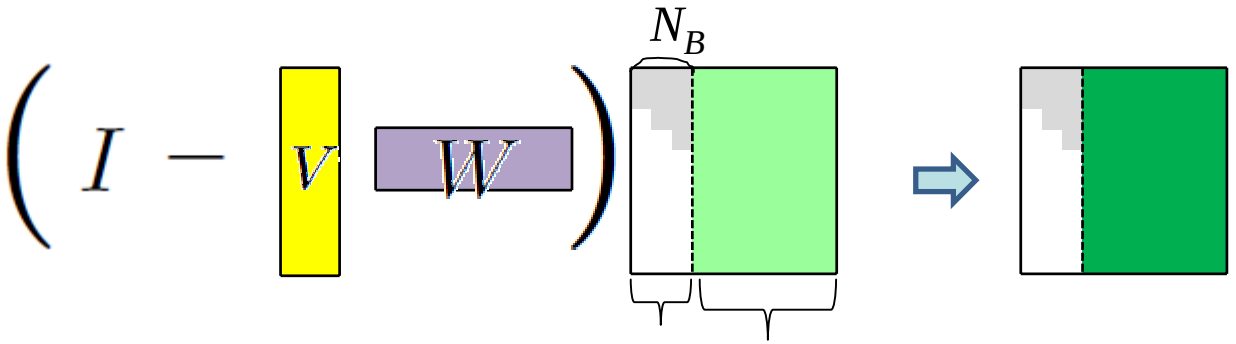
近年のCPUは (演算速度) ≫ (メモリアクセス速度)

⇨ **メモリアクセス速度がネックに**

ハウスホルダー変換のブロック化

- 複数のハウスホルダー変換を1つに合成

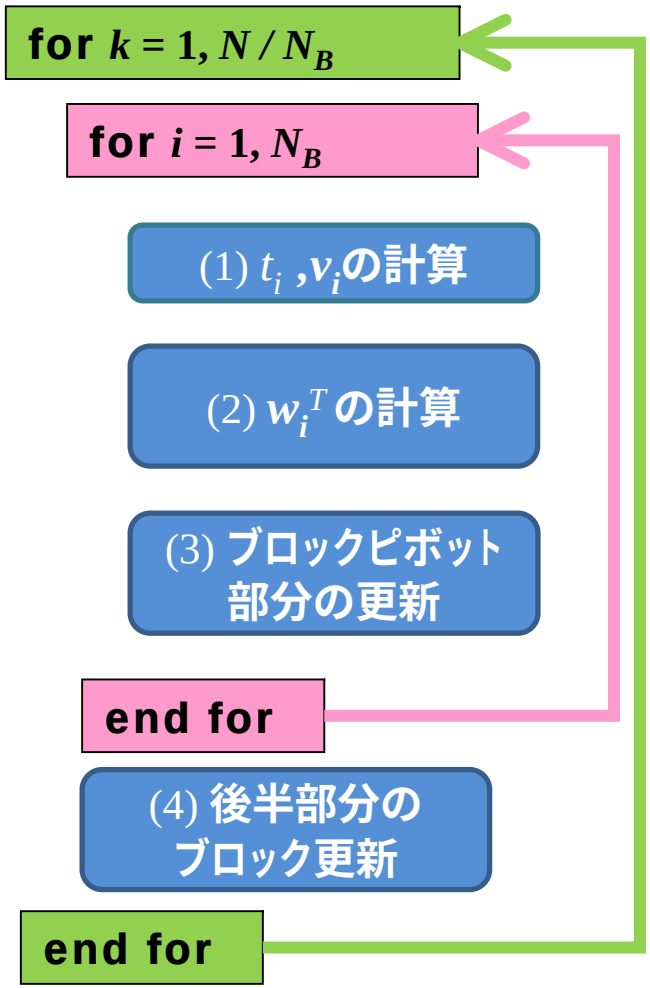
$$(I - t_{N_B} v_{N_B} v_{N_B}^T) \cdots (I - t_2 v_2 v_2^T)(I - t_1 v_1 v_1^T) \Rightarrow I - V W$$

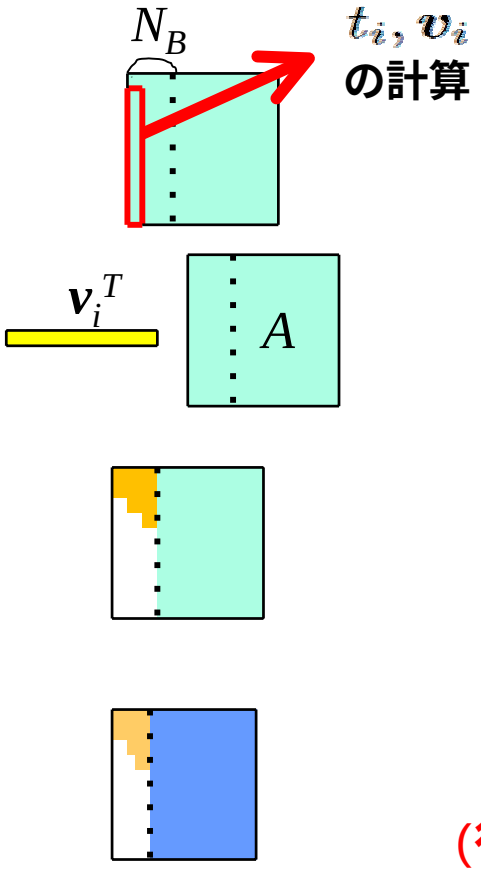


行列の更新の部分を，効率のよい行列乗算で行える

データアクセス回数 $O(N^2)$, **演算量** $O(N^2 \cdot N_B)$

ハウスホルダー変換のブロック化 (続き)





計算量

$O(N)$

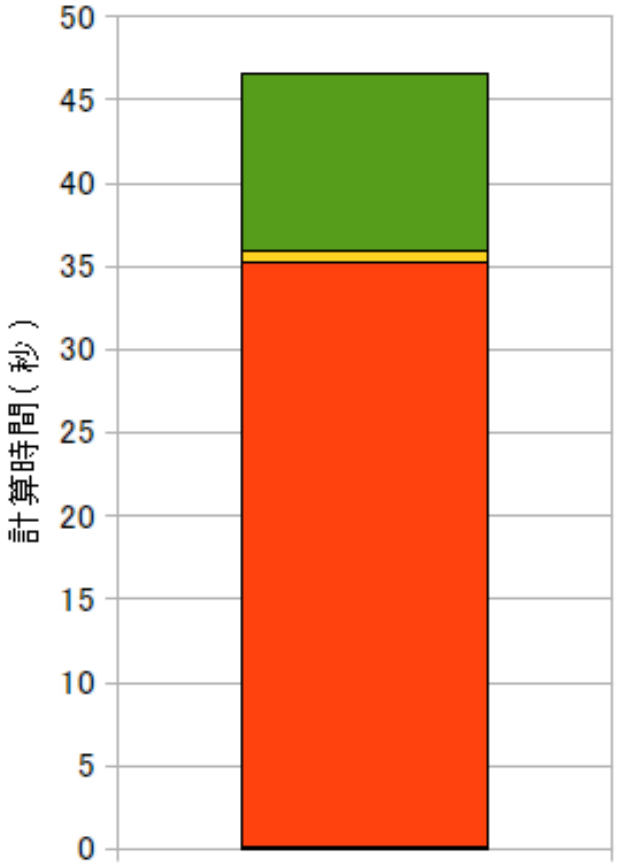
$O(N^2)$

$O(N \cdot N_B)$

$O(N^2 \cdot N_B)$
(行列乗算で実行可能)

ハウスホルダー変換のブロック化 (続き)

- ブロック化アルゴリズムの計算時間の内訳



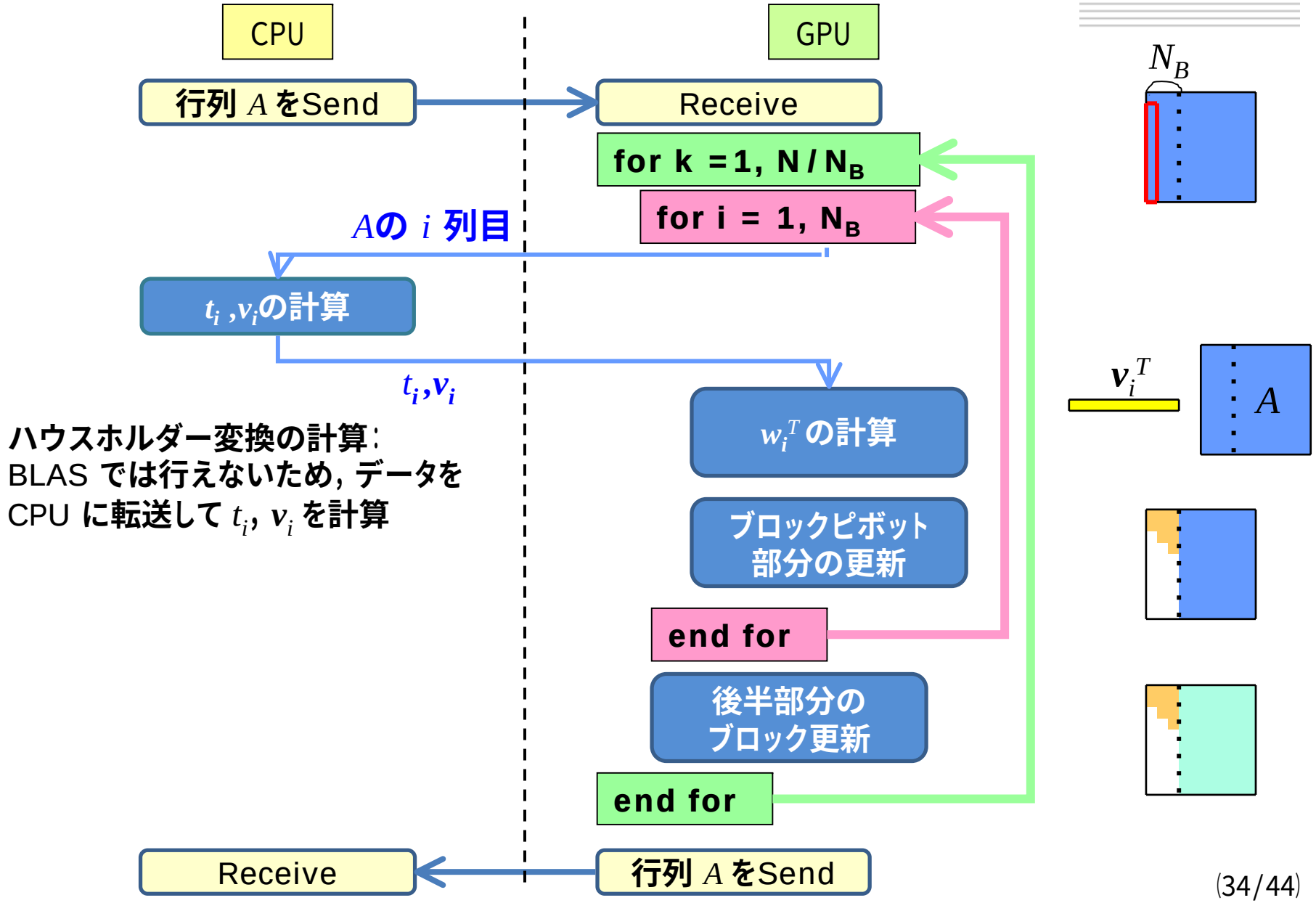
計算機環境
CPU : Core i7 920 (2.66 GHz) [4コア使用]
Memory : 6.0GB
N = 5120, 倍精度

- (4)後半部分に対するブロック更新
- (3)ブロックピボット部分の更新
- (2) $\boldsymbol{w}_i^T = t_i \boldsymbol{v}_i^T A$
- (1) t_i, $\boldsymbol{v}_i$ の計算

ブロック化アルゴリズムでも，行列ベクトル積が性能のボトルネック

⇨ GPU の高いメモリアクセス性能を利用して高速化

実装方式(a)：BLAS をすべて GPU で実行



ハウスホルダー変換の計算：
BLAS では行えないため，データを
CPU に転送して t_i, v_i を計算

実装方式(a)の性能評価

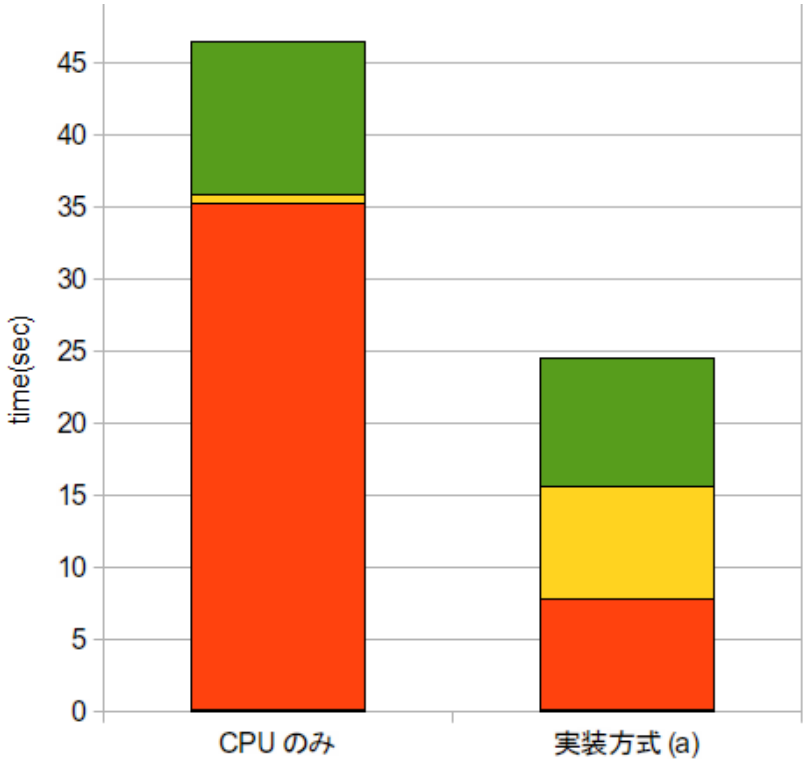
- テスト行列
  - 倍精度乱数行列 ($N = 5120$)

- 計算機環境

CPU	Core i7 920 (2.66 GHz) [4コア使用]
CPU Memory	6.0GB
コンパイラ	gcc ver.4.1.2 (オプション-O3) Intel Fortran コンパイラ ver.9.1(オプション-O3)
CPU LAPACK	Intel Math Kernel Library 11.0
GPU	Tesla C1060
GPU Memory	4.0GB
CUDA version	2.0

実装方式(a)の性能評価

- ヘッセンベルグ化の各部分の演算時間 $(N = 5120)$

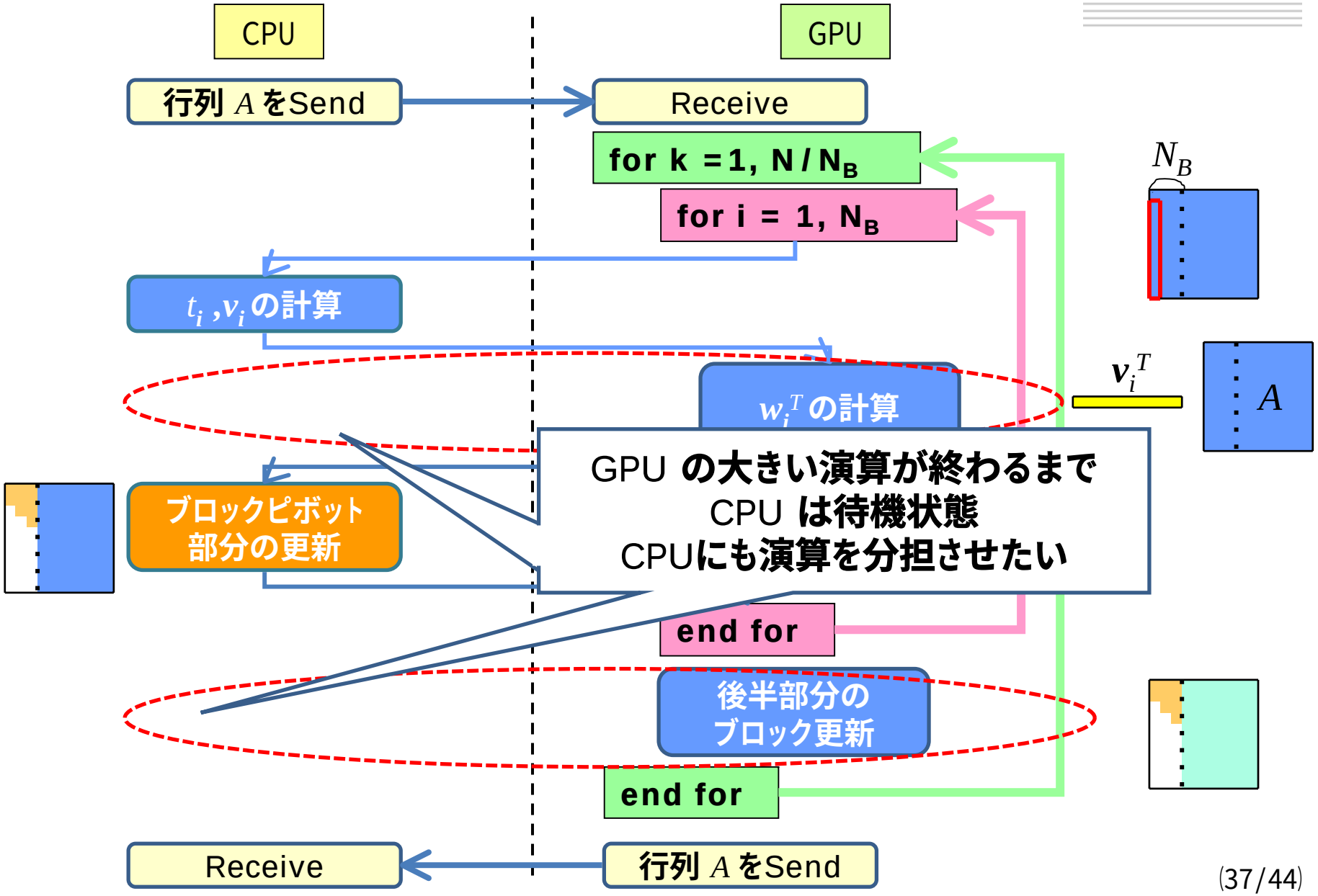


- (4)後半部分に対するブロック更新 $O(N^2 \cdot N_B)$
- (3)ブロックピボット部分の更新 $O(N \cdot N_B)$
- (2) $\boldsymbol{w}_i^T = t_i \boldsymbol{v}_i^T A$ $O(N^2)$
- (1) t_i, $\boldsymbol{v}_i$ の計算 $O(N)$

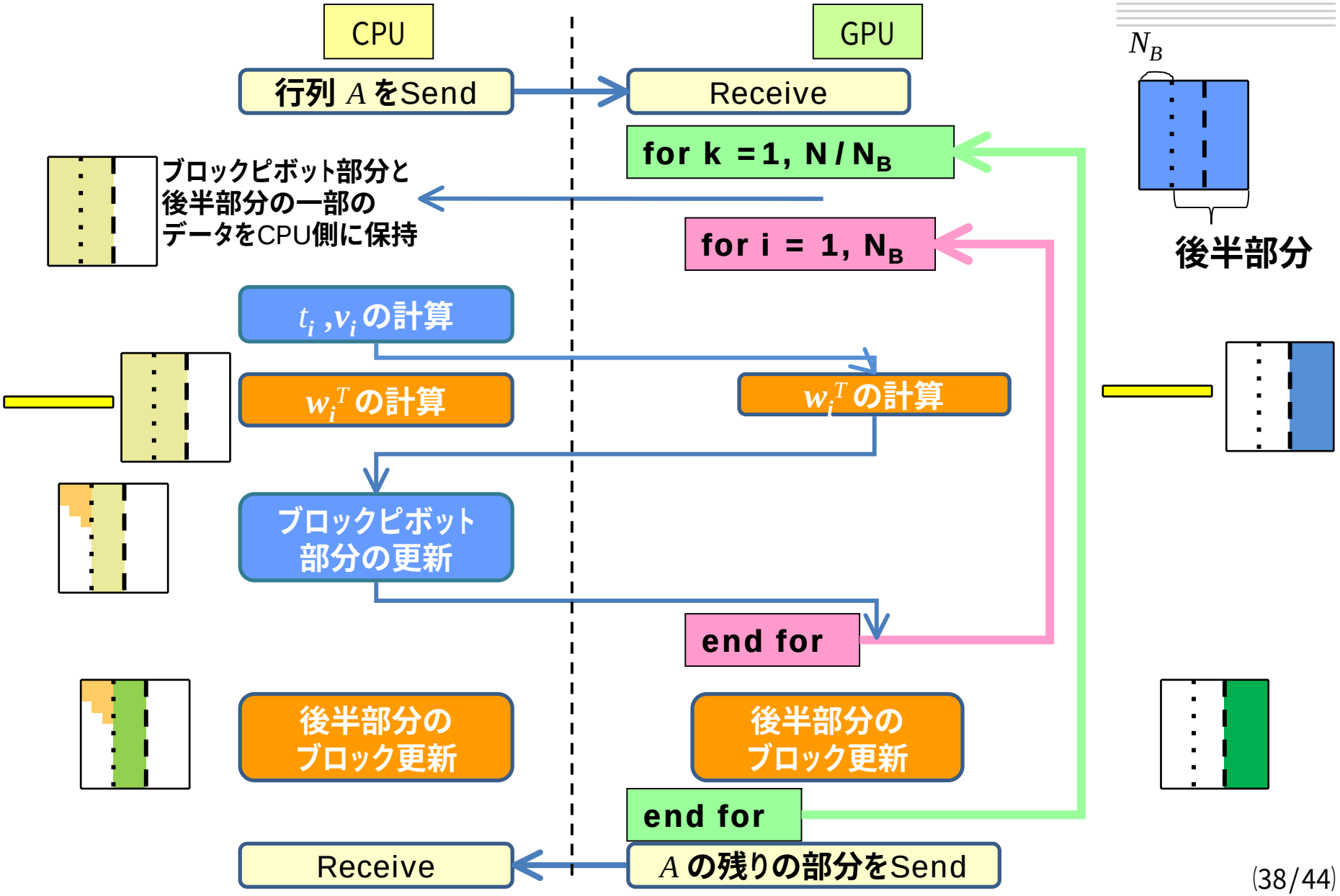
(2)は高速化できたが, (3)の時間は逆に増大

⇨ 演算量の小さい BLASでの GPU の性能低下が問題

実装方式(b)：小サイズの BLAS は CPU で実行



実装方式(c)：CPU と GPU の負荷分散



実装方式のまとめ

	実装方式(a)	実装方式(b)	実装方式(c)
方針	すべてのBLASをGPUで実行	小サイズのBLASをCPUで実行	大サイズのBLASをCPU･GPUで協調して実行
(1) t_i ,v_i の計算	CPU	CPU	CPU
(2) w_i^T の計算	GPU	GPU →	CPU＋GPU
(3)ブロックピボット部分の更新	GPU →	CPU	CPU
(4)行列の後半部分の更新	GPU	GPU →	CPU＋GPU

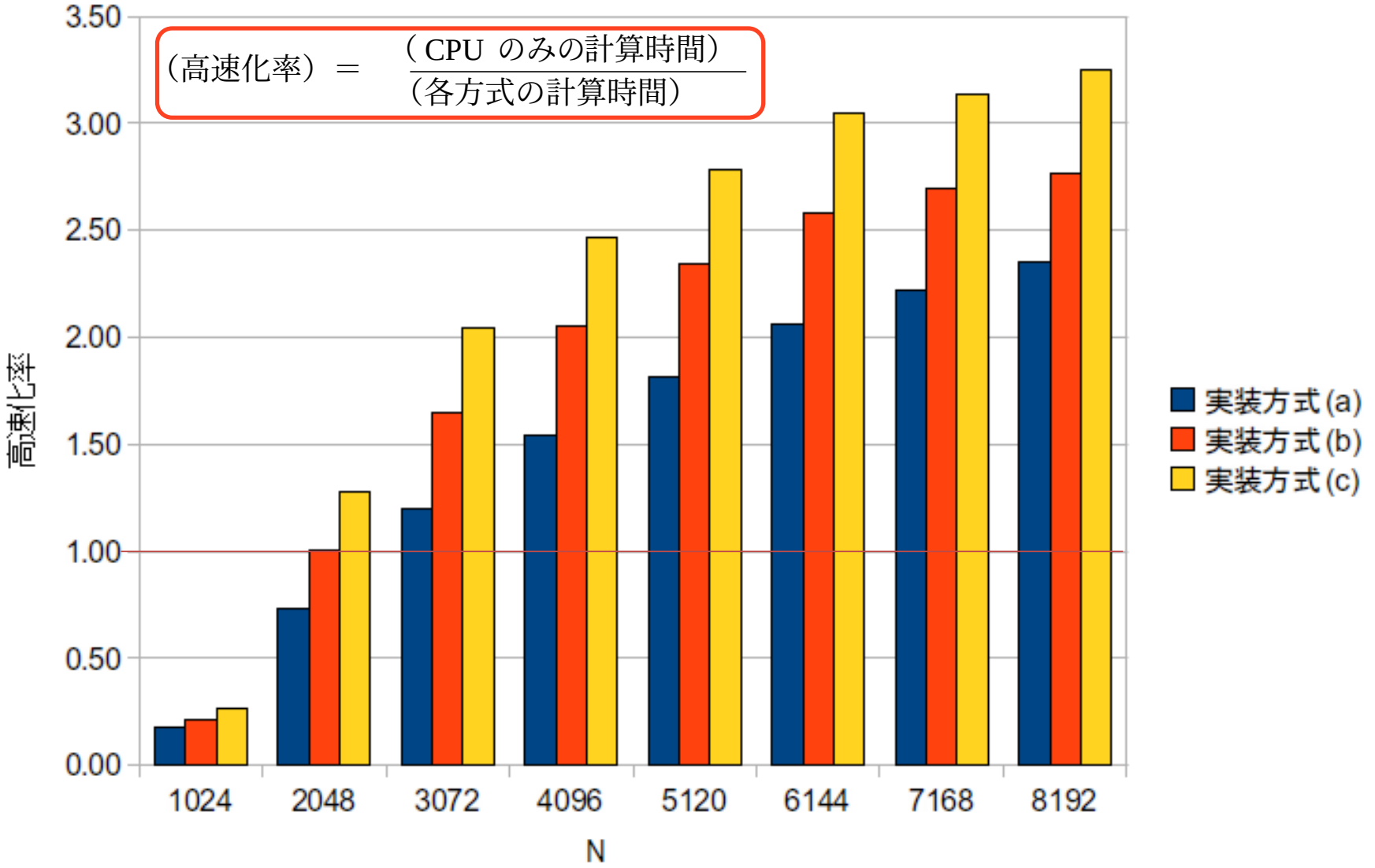
性能評価

- **テスト行列**
  - **倍精度乱数行列** A $(N = 1024, 2048, \ldots, 8192)$
- **評価対象**
  - **次の**4**つの方式でヘッセンベルグ化を行い，実行時間を計測**
    - CPU **のみ** (4**コア使用**)
    - **実装方式**(a), (b), (c)
  - **高速化率を求める**
- **評価条件**
  - **ブロックサイズ**：$N_B = 32$ **で固定**
  - **実装方式**(c)**で**CPU **データ比率**

} **予備実験から求めた最適値**

N	1024	2048	3072	4096	5120	6144	7168	8192
CPU**比率**	24/32	10/32	8/32	6/32	5/32	5/32	5/32	5/32

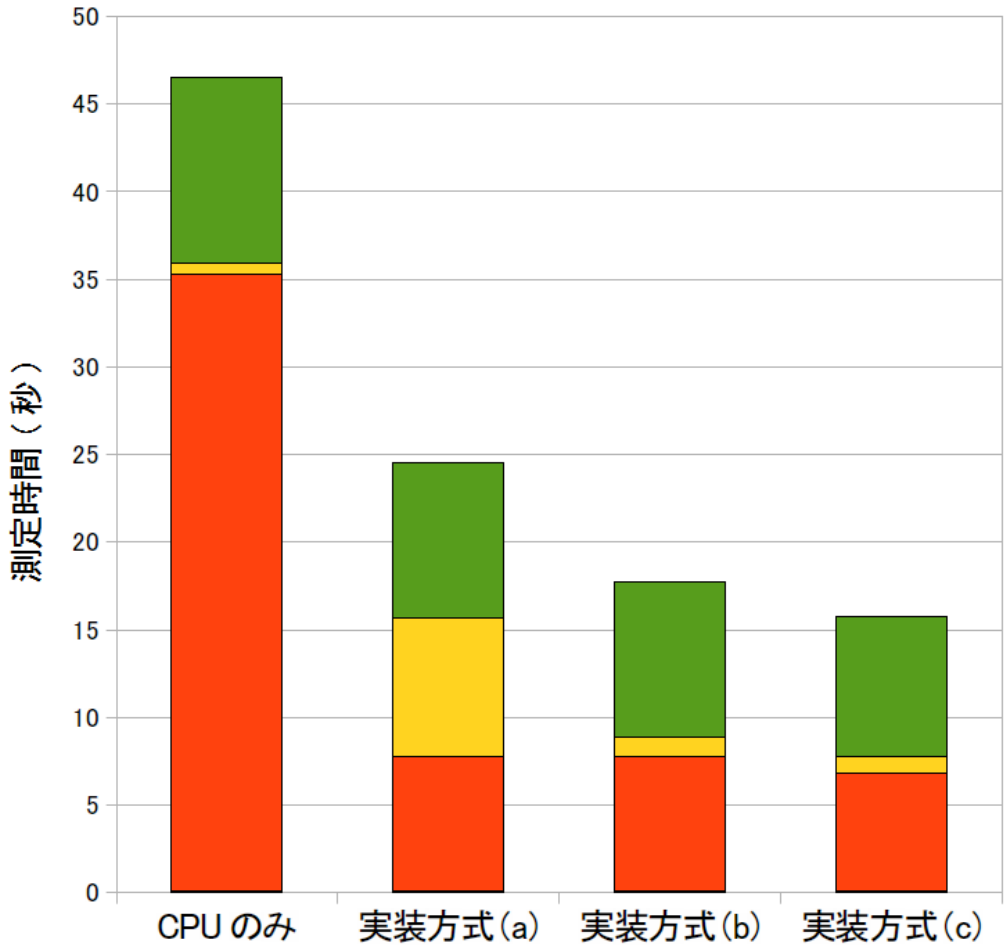
各実装方式の高速化率



実装方式(c)は N＝8192 のとき3.25倍の高速化を達成

各実装方式の実行時間

- ヘッセンベルグ化の各部分の演算時間 $(N = 5120)$



(4)後半部分に対するブロック更新

(3)ブロックピボット部分の更新

(2) $\boldsymbol{w}_i^T = t_i \boldsymbol{v}_i^T A$

(1) t_i, $\boldsymbol{v}_i$ の計算

まとめ

- 非対称密行列の固有値計算で多くの時間を占めるヘッセンベルグ化を，GPU の高いメモリ転送性能を利用して高速化した。GPU の使用に当たっては，CUBLAS を利用した

- 性能評価の結果，小さいサイズの BLAS は CPU で実行し，大きいサイズの BLAS は CPU と GPU で分担して実行するのが最も効果的であった

- 上記の実装方針を用い，負荷分散を最適化した結果，Tesla C1060 では Core i7 (4コア使用) に比べて最大で3.25倍の高速化が得られた

今後の課題

- より高性能な BLAS ライブラリ (MAGMA) の組み込みと性能評価
- 非対称固有値計算の他の部分への GPU の適用
- 性能モデリングに基づく負荷分散の自動最適化