CPU/GPU ヘテロジニアスクラスタのための のタイル行列分解の研究

山梨大学大学院
医工農学総合教育部
博士課程学位論文

2021 年 3 月
高柳雅俊

目 次

図 目 次

表 目 次

第1章　序論

1.1　目的

シミュレーションや設計で行われる科学技術計算では，ベクトルや行列の形式でデータを扱う．行列の上・下三角化，対角化などの基本的な操作や固有値分解，特異値分解，線形方程式の求解などは数値線形代数と呼ばれ，科学技術計算分野の基礎領域の広い部分を占める．数値線形代数計算の 1 つに QR 分解があり，固有値分解や特異値分解の前処理や後処理などに多用される計算である．QR 分解の計算には，ギブンス回転やグラムシュミット直交化，ハウスホルダー変換を用いた手法があるが，数値的安定性や計算速度の面からハウスホルダー変換による QR 分解がよく用いられる．

高速化手法の 1 つであるブロックアルゴリズムを用いた QR 分解は，ブロック QR 分解と呼ばれている．QR 分解の逐次アルゴリズムでは行列ベクトル積が主要な演算であるのに対し，ブロック QR 分解では行列・行列積が主要演算となる．行列・行列積は，データ参照の局所性を高めることでキャッシュの有効活用が可能であり，大幅な高速化が可能となる．そのため，ブロック QR 分解は，メモリ階層を持つアーキテクチャ向けに最適化することが可能であり，高いパフォーマンスを発揮するため広く用いられている．行列・行列積は並列性も高いため，並列コンピュータにおいても高いパフォーマンスを発揮する．ブロック QR 分解は行列・行列積による計算を多用する事で高速化実装されているが，逐次実行の部分もある．このような並列計算モデルは fork-join モデルと呼ばれる．fork-join モデルに代表される Bulk Synchronous Parallel モデルは並列化が容易であるが，並列計算資源のすべてを有効に活用することは原理的に不可能であり，近年の高並列なコンピュータのために別のパラダイムの普及が望まれている．

Buttari と Langou は数値線形代数計算の並列化に対して，マルチコアアーキテクチャ向けの新しいアプローチを行った [1]．これは，行列を小行列（タイル）に分割し，1 または 2 タイル毎に処理を行うタイルアルゴリズムを用いることで，fork-join モデルではなく，タスク並列による行列分解の高並列化を行うものだった．タイルアルゴリズムは，タイルサイズを適切に選択することで並列計算資源の量に対して十分な数のタスク数を確保し，これらを非同期に実行することですべての計算資源を休みなく動作させることを目的としており，近年の主流であるマルチコアアーキテクチャ向けアルゴリズムである．

図 1.1 にブロックアルゴリズムとタイルアルゴリズムによる QR 分解の実行時間を示す．ブロックアルゴリズムの実装は netlib で公開されている LAPACK 3.8.0[2] の dgeqr，BLAS ライブラリには Intel Math Kernel Library (MKL) 2018.0.128 を使用した．タイルアルゴリズムの実装は PLASMA 2.8.0[3] のものを使用し，BLAS ライブラリには openBLAS 0.2.20[4] を使用した．実験には，Xeon E5-2640 v2 を使用した．図 1.1 から，タイル QR 分解はブロック QR 分解よりも高速であることが分かる．また，図 1.2，1.3 は行列サイズを固定し，計算コア数を変更した時の実行時間および計算速度のグラフである．コア数を増加させた時，タイルアルゴリズムがブロックアルゴリズムと比較して性能向上しているのが確認できる．

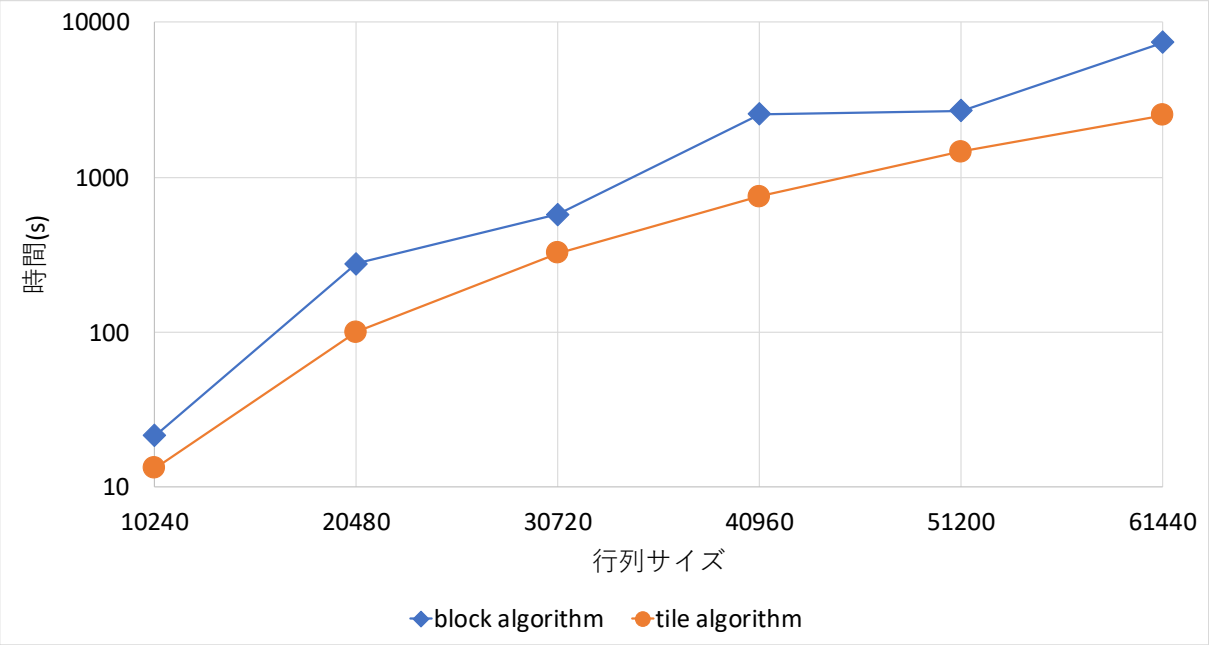

図 1.1: 各 QR 分解手法の実行時間．正方行列に対する QR 分解を 5 回計測を行い，最大・最小を除いた 3 点の平均実行時間．行列の値はメルセンヌ・ツイスター法による乱数から生成．横軸は正方行列の行列サイズ，縦軸は実行時間の対数軸．

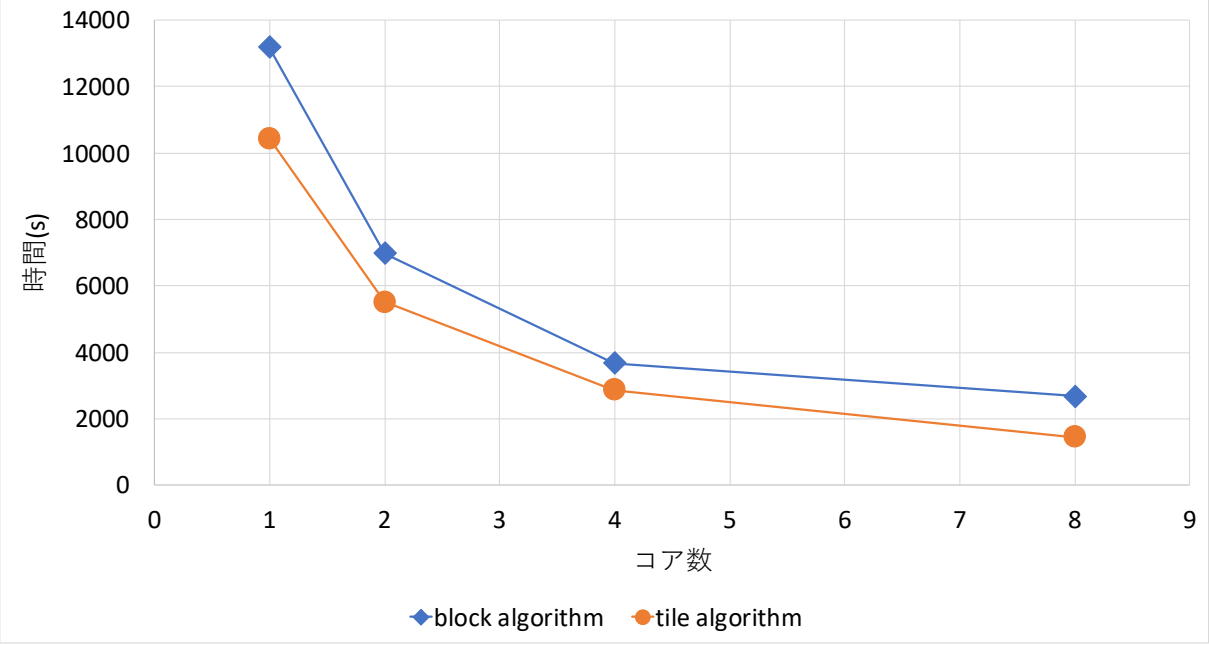

図 1.2: 各 QR 分解手法の実行時間．行列サイズ 51200 に対して QR 分解を 5 回計測を行い，最大・最小を除いた 3 点の平均実行時間．行列の値はメルセンヌ・ツイスター法による乱数から生成．横軸は使用コア数，縦軸は実行時間．

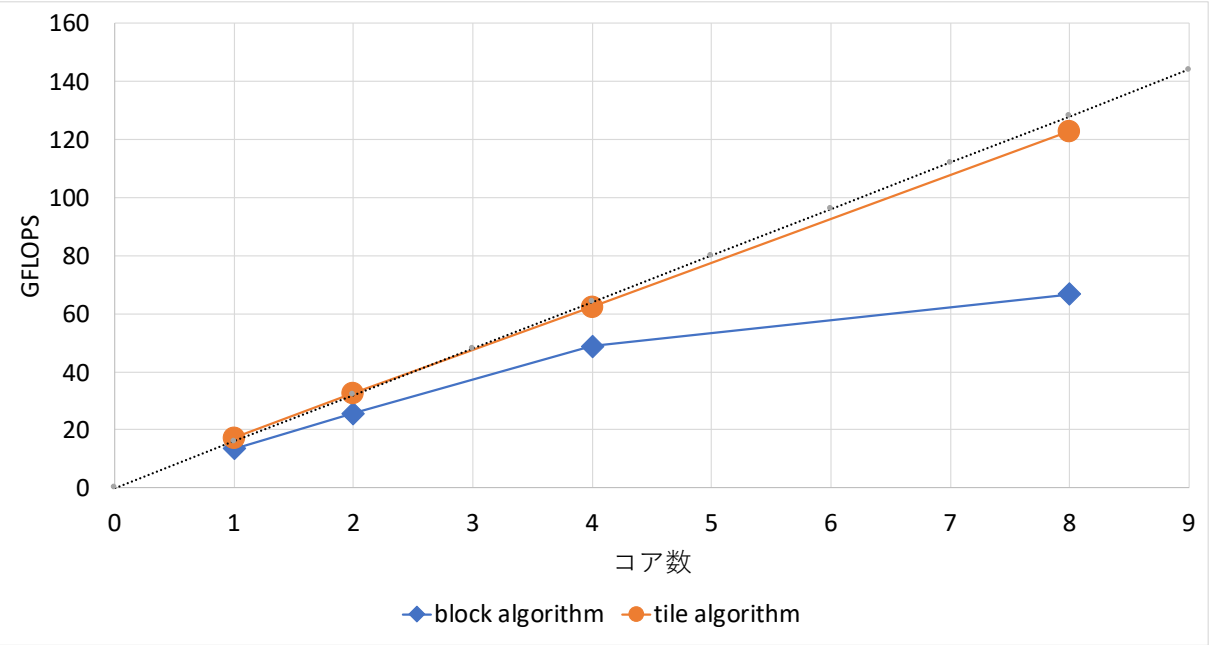

図 1.3: 各 QR 分解手法の演算性能．行列サイズ 51200 に対する QR 分解の平均実行時間から，QR 分解の演算量は行列サイズを n とした時，$4/3n^3$ とした．行列の値はメルセンヌ・ツイスター法による乱数から生成．横軸は使用コア数，縦軸は計算速度 GFLOPS.

一方で，前述のとおり，タイルアルゴリズムは適切なタイルサイズを選択しなければ高い性能を発揮できない．図 1.4 は行列サイズを固定した時のタイルサイズ，内部ブロック幅変更時の実行時間である．ここで内部ブロック幅とは，タイルアルゴリズム内で使用されるブロックアルゴリズムにおけるブロック幅である．図 1.4 から読み取れるように，タイルサイズによっては明確な実行時間差が現れる．適切なタイルサイズを選択するためにはパラメータ探索を行う必要があるが，パラメータ空間の全探索を行うと行列サイズによっては，3〜4 日またはそれ以上に探索時間が必要となり，計算資源の有効活用という点で大きな問題となっている．この問題に対して，効果的な枝刈りによってパラメータ探索を高速に行う手法と，性能モデルを構築してパラメータ設定を行う手法が考えられる．本研究では，性能モデル構築によるパラメータチューニングを行った．

また，近年のコンピュータ環境は大きく変化している．図 1.5 に 2020 年 11 月の TOP500 ランキング [5] のスーパーコンピュータにおけるシステム全体の計算コア数のグラフを示す．横軸は TOP500 のランキング，縦軸は計算コア数を表わす．TOP500 はスーパーコンピュータの計算速度ランキングであり，毎年 6 月と 11 月に発表される．TOP500 にランクインしている並列計算マシンのコア数は少なくとも 10000 以上，最大のものでは 1000 万コアにもなっている．図 1.5 より，GPU を搭載したスーパーコンピュータが TOP500 で約 1/3 を占め，それ以外の演算加速装置 (Others) も多く存在することが分かる．2008 年 11 月のランキングで，東京工業大学の TSUBAME 1.2 は NVIDIA 社の GPU を搭載したスーパーコンピュータとして初めて上位（30 位）にランクインした．その後，演算加速装置として GPU を搭載したスーパーコンピュータのランクインは増加し，最新のランキングで 2 位となった Summit は演算性能の約 90%が GPU によるものだとされており，GPU を汎用的な計算に用いる GPGPU(General-purpose computing on graphics processing units) はスーパーコンピュータの分野に定着している．

このように，近年の科学技術計算に用いられるコンピュータの並列性は非常に高く，システムは複雑化している．これらのコンピュータに適した数値計算ライブラリが求められ

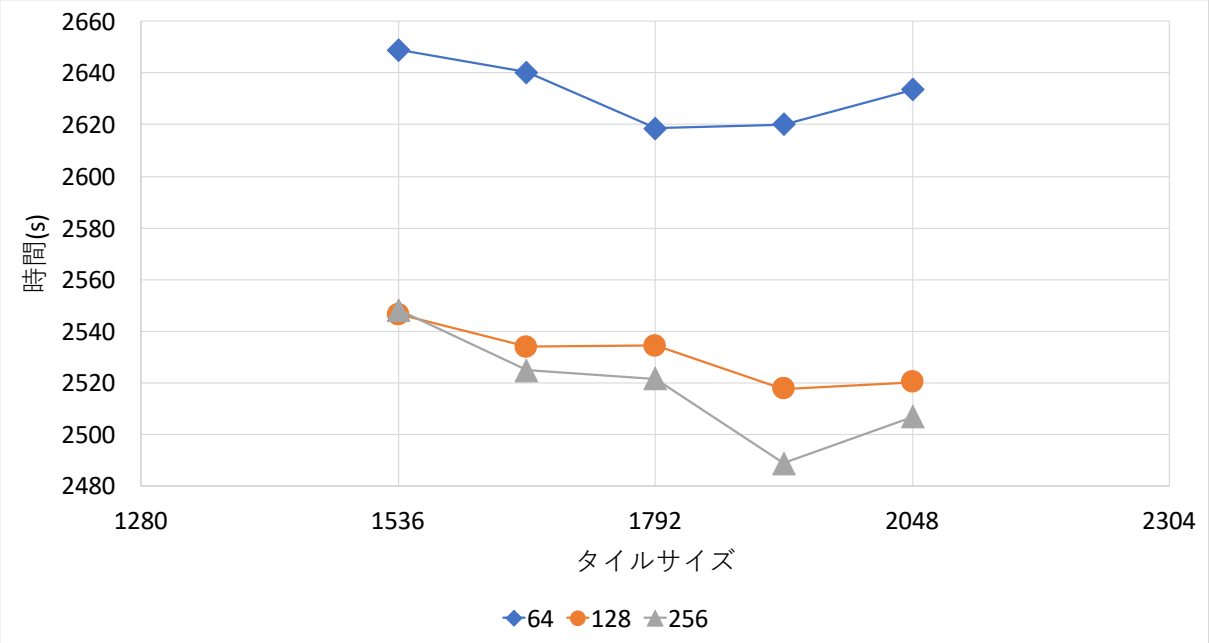

図 1.4: タイル QR 分解手法の演算性能．行列サイズ 61440 固定した時の内部ブロック幅，タイルサイズ変更時の実行時間．行列の値はメルセンヌ・ツイスター法による乱数から生成．横軸はタイルサイズ，縦軸は実行時間．

ている．

以上を踏まえて，本研究では 2 つの課題に取り組む．1 つはマルチコアクラスタシステムにおける最適なタイルサイズ選択のための性能モデル作成である．2 つ目は，CPU/GPU ヘテロジニアスクラスタシステムに適した高い計算速度を有する数値線形代数ライブラリのための QR 分解の高性能実装である．

1.2 成果

本研究の成果の一つ目は，マルチコアクラスタシステムにおける性能モデルを作成することで実行時間の予測を行えることである．タイルアルゴリズムのタスク並列モデルにおける各タスクは，1 または 2 タイル毎の処理であり，この処理を行う小プログラムをカーネルと呼ぶ．本研究で作成した性能モデルは，3 種類のカーネルのモデルを含んでおり，各タイルサイズについて，この 3 種類のカーネルの実行時間を求めるだけで，誤差約 10%で実行時間を求めることが可能となった．

二つ目の成果は，CPU/GPU ヘテロジニアスクラスタシステムにおけるタイルアルゴリズムの実装と性能評価である．これまで，大規模並列環境における CPU/GPU クラスタシステムのための数値計算ライブラリの整備は行われていない．タイルアルゴリズムは高並列な計算資源を有効活用できる手法であるが，分散メモリ環境では小規模な通信を大量に発生させてしまうため，通信回数を削減する工夫が必要となる．また，本研究では GPU と CPU の両方を計算資源として効果的に活用することを目指す．CPU と GPU の特性の違いからそれぞれに分解タスクと更新タスクを割り当てることにした．さらに本研究では，更新タスクの実行方法について 2 種類の手法を提案，実装を行い，実験の結果から並列性，計算と通信のオーバーラップの点から Stream Update と呼ばれる手法が有効であることが分かった．

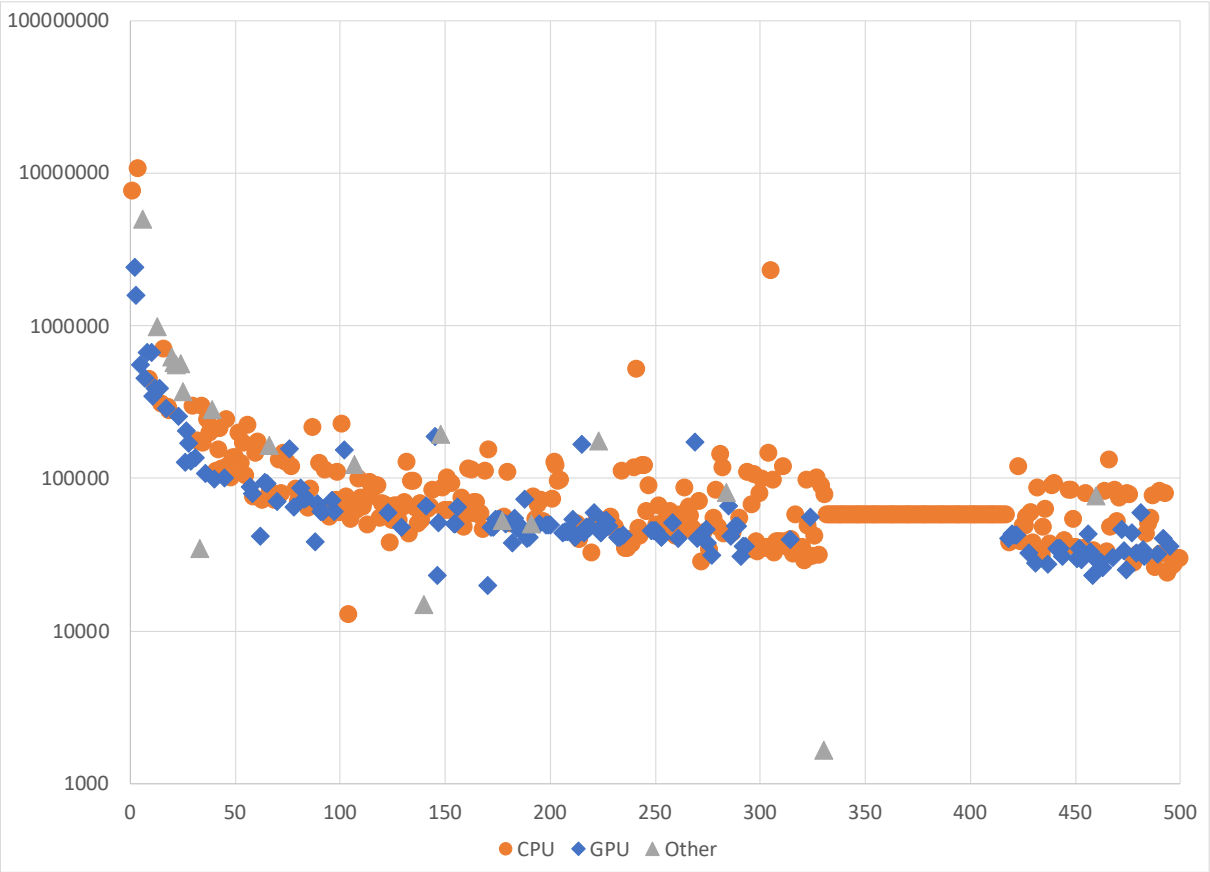

図 1.5: 2020 年 11 月の TOP500 ランキングシステムのコア数

1.3 論文構成

はじめに，研究の背景，成果について述べた．

2 章については研究の背景，現在および今後の大規模並列環境について，また，数値計算ライブラリの必要性について示す．

3 章では並列化プログラムの実装に当たって必要となる技術，MPI や OpenMP について紹介する．

4 章では本研究で取り上げる QR 分解について，特にブロックアルゴリズムやタイルアルゴリズムなどの並列化手法について示す．

5 章では大規模並列下における通信量を減らすためのアルゴリズム，Communication Avoiding について述べる．

6 章ではタイルアルゴリズムにおける重要なパラメータ，タイルサイズの最適値探索手法について述べる．

7 章では本研究で行った CPU/GPU ヘテロジニアス環境におけるタイル QR 分解の 2 種類の計算手法について述べる．

8 章では今回行った実験について考察を行い，9 章で本論文を締める．

第2章　研究背景

2.1　並列プログラミングの必要性

図 2.1 に 1970 年代から 2013 年までの CPU のトランジスタ数の変化を示す [6].「半導体の集積率は 18ヶ月で 2 倍になる」というムーアの法則に即した形でこの期間の CPU のトランジスタ数は着実に増加していた．図 2.2 に CPU のクロック周波数の変化を示す [6]．これを見ると，クロック周波数は 2005 年以前と，2005 年から 2013 年の間では全く異なる傾向を示し，増加傾向から停滞に転じている．図 2.3 に CPU に搭載されたコア数と実行可能なスレッド数の変化を示す [6]．2005 年頃から CPU のコア数が急激に増加し始めたことが分かる．

1971 年に Intel 4004 CPU が発売されて以来，トランジスタ数の増大とクロック周波数の上昇によって CPU は爆発的にその性能を向上させていた．一方，CPU の消費電力もまた，クロック周波数に比例する形で増大した．図 2.4 に 1980 年代後半から 2010 年までの CPU の消費電力の変化を示す [7]．この図から 2000 年代前半には 100W を越える消費電力の CPU が存在していることが分かる．

消費電力が大きい CPU は発熱量も多く，これを冷却するための装置が必要となり，コンピュータの総消費電力は更に大きくなる．CPU 性能はそのトランジスタ数とクロックに依存する．消費電力の増大により，CPU のクロック周波数を上げられなくなり性能向上が停滞してしまう問題は「The Power Wall（電力の壁）」と呼ばれる．2005 年までのクロック周波数の上昇は電力の壁によって頭打ちとなった．

2005 年に世界初の 2 コアのプロセッサ Intel Pentium D プロセッサが発売された．これまでクロック周波数高くすることで得られていた性能向上を，プロセッサコアを増やすことで得ようとする方向への転換であった．図 2.2，2.3 から，これ以降クロック周波数の増加はほとんどみられず，コア数が指数的に増加していくことが分かる．図 2.1，2.4 から，2005 年以降もトランジスタ数はムーアの法則に従い増加し続けているが，消費電力の増加はわずかであることが分かる．

2005 年に発売された雑誌に「The Free Lunch is Over」という記事が掲載された [8]．クロック周波数の向上とともに CPU 性能が向上していた 2005 年まではソウフトウェア技術者は開発した製品の改良をしなくてもよかったが，2005 年以降は CPU コア単体での性能向上はほとんどないので，複数の CPU コアを効果的に活用する並列プログラミング技術を導入しソフトウェア開発をしなければならなくなった．[8] ではこのことが指摘されている．

コンピュータシミュレーションを活用した計算科学が，理論，実験に続く第 3 の科学に対するアプローチであるとされて久しく，2013 年のノーベル化学賞は巨大分子の化学反応のシミュレーションが可能なマルチスケールモデルを開発した研究者が受賞した．コンピュータの性能向上に伴って計算科学は様々な分野に広がり，より正確なシミュレーションを実施するためにより大規模な問題を解くことが求められている．2005 年から始まった CPU コアの増加は現在も続いており，最新のコンシューマ向け CPU である AMD Ryzen Threadripper 3990X は 64 コア（128 スレッド）を搭載している．ムーアの法則が終焉す

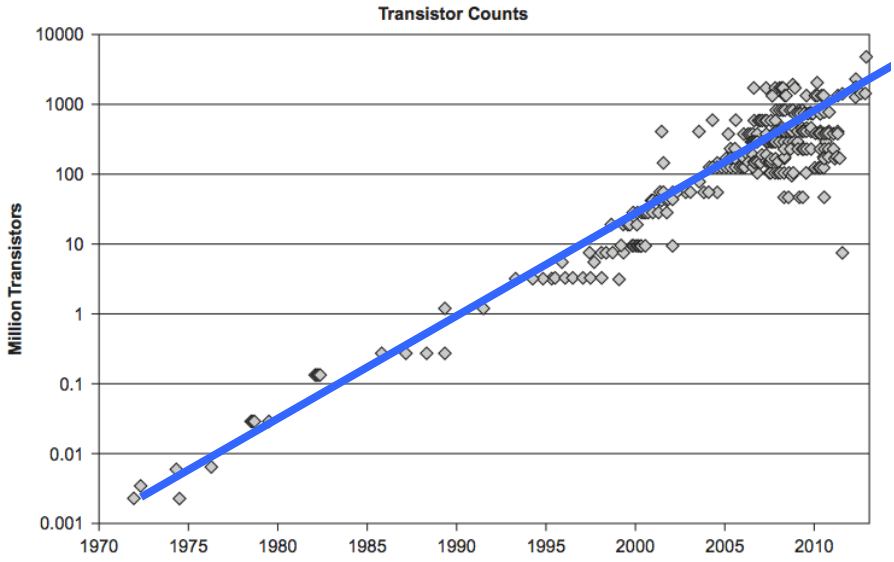

図 2.1: CPU のトランジスタ数の経年変化

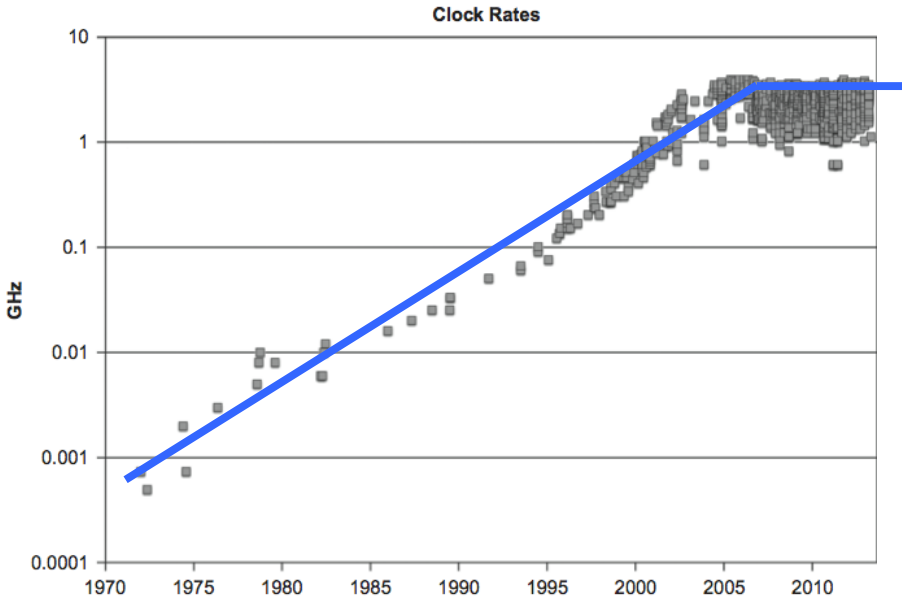

図 2.2: CPU のクロック周波数の経年変化

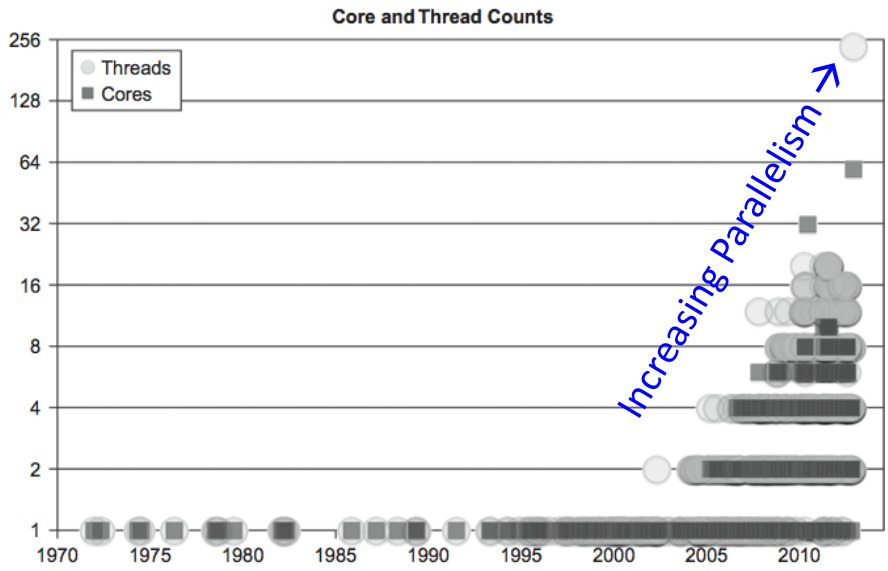

図 2.3: CPU のコア数と実行可能スレッド数の経年変化

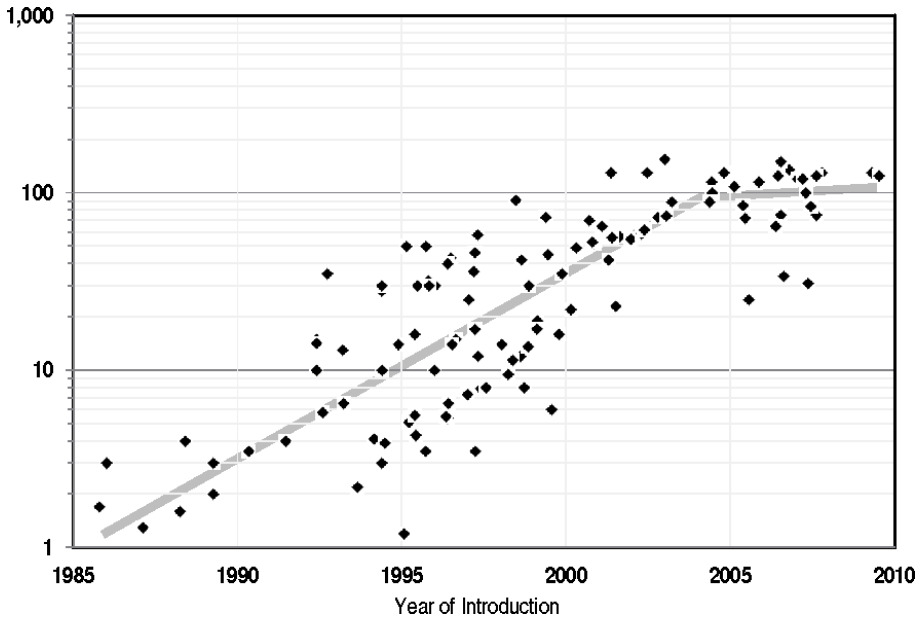

図 2.4: CPU の消費電力の経年変化．縦軸は消費電力（W）

ると予測される 2020 年代後半まではCPU コアの増加傾向が続くと予想できる．このようなメニーコアアーキテクチャのすべての計算資源を効率的に活用することで，大規模な科学技術計算を高速に実行することが現在，求められている．

2.2 高並列環境のトレンド

ここでは前章でふれたTOP500 ランキングの上位のスーパーコンピュータについて追記する．

コンピュータは誕生から現在までさまざまな工夫により演算性能の向上が行われてきた．CPU のクロック周波数の向上，Layer-1 キャッシュやLayer-2 キャッシュの導入による，メモリとレジスタ間のボトルネック解消や，CPU 内に複数の計算コアをパッケージ化するマルチコア CPU などにより性能向上し続けている．性能の向上に伴い，さまざまな科学技術計算がコンピュータシステムで行われ，また，対象とする問題範囲を拡大し続けてきた．なかでも，大規模な科学技術計算に用いられるスーパーコンピュータはパーソナルコンピュータと比較して数十万倍の演算速度を持ち，大規模科学技術計算の発展を支えてきた．

表 2.1: TOP500 ランキング上位 10 位（2020 年 11 月）

Rank	Name	Country	Total Cores	Accelel. Cores	Rmax [Pflop/s]
1	Supercomputer Fugaku	Japan	7,630,848		442
2	Summit	US	2,414,592	2,211,840	149
3	Sierra	US	1,572,480	1,382,400	95
4	Sunway TaihuLight	China	10,649,600		93
5	Selene	US	555,520	483,840	63
6	Tianhe-2A	China	4,981,760	4,554,752	62
7	JUWELS Booster Module	Garmany	449,280	404,352	44
8	HPC5	Italy	669,760	582,400	35
9	Frontera	US	448,448		24
10	Dammam-7	Saudi Arabia	672,520	632,960	22

2020 年 11 月における TOP500 ランキングの上位 10 機種を表 2.1 に示す．一位は理化学研究所が所有する富岳であり，442PFLOPS の演算能力を有している．また，二位はDOE/SC/Oak Ridge National Laboratory の Summit であり，149PFLOPS の演算性能を有しており，高い性能を得るために Graphics Processing Unit（GPU）を主要な演算装置として使用している．

初期の GPU は主としてゲームや映像の高画質化のためのグラフィックス処理を行うために用いられていた．GPU は内部にプログラマブルシェーダと呼ばれる演算装置を多数所有していたため，高い並列性と演算性能を持っている．その演算性能に着目し，OpenGL や DirectX といった API を通して，グラフィックス処理以外のさまざまな計算を行う試みが始まった．これらの技術を用いたプログラミングは専門性が高く容易ではなかったが，NVIDIA 社から GPU のための統合開発環境，CUDA が提供されたことにより，GPU を用いたプログラム開発は大きく改善された．CUDA が提供されて以降， GPU アーキテクチャは数年おきに見直され，多数の計算コアやメモリ，キャッシュ，テンソル計算のため

の Tensor Core，高速な専用通信路の導入などにより，GPU はより高性能な演算装置へと発展し続けている．

GPU は，CPU と比較すると性能は劣るが多数の演算処理装置を内部に搭載しており，非常に高い浮動小数点演算能力を持っている．ただし，GPU で計算を行うには，GPU メモリにデータを配置しなければならないため CPU-GPU 間のデータ移動が必要であり，このメモリは高速であるがメインメモリと比べて容量が小さい．

現在では CUDA 以外にも，OpenACC，OpenCL など GPU の種類を問わない GPU プログラミング環境が多数提供されており，GPU を用いた科学技術計算，音声処理，機械学習などの汎用的なプログラミングが広く行われるようになった．本研究では，NVIDIA 社から提供されている高速な BLAS ライブラリ，cuBLAS を使用するため，OpenACC，OpenCL の採用を見送った．GPU を使用した高性能計算は今後さらに拡大すると予想できる．

これまで述べたように，演算加速装置として GPU を用いた大規模な並列システムが普及している一方で，科学技術計算で使われる数値計算ライブラリはまだ GPU を有するシステムに対して最適化がされていない．

2.3 数値計算ライブラリの歴史

スーパーコンピュータによって，様々な科学技術計算が行われるようになり，数百，数千個の CPU を用いた並列計算が日常的に行われる時代となった．このような大規模並列計算を支援するための基盤ソフトウェアが科学技術計算では求められている．

科学技術計算分野では，様々な計算の中でもベクトルや行列の基本演算を行うライブラリに BLAS(Basic Linear Algebra Subprograms) がある．BLAS では，ベクトルや行列の加減算，スカラー倍，積などの基本的な演算が Level 1，Level 2，Level 3 と 3 種類に分かれている．Level 1 はベクトル同士の演算，Level 2 は行列ベクトル演算，Level 3 では行列同士の演算が定義されている．近年のコンピュータはメモリアクセス性能に対して演算性能が非常に高い．そのため，メモリへのデータアクセスが全体の演算性能のボトルネックとなっている．表 2.2 に各 Level のメモリアクセス回数と演算回数を載せる．Level が上がるにつれてデータ参照回数に対する演算回数は増加している．このため，高速な演算には Level 3 BLAS を使用する事が望ましい．また，BLAS で実装された演算を使用して，より

表 2.2: 各 Level のメモリアクセスと演算比 (n:データ量)

Level	メモリ参照回数 (M)	演算回数 (F)	M:F 比
Level 1	3n	2n	3:2
Level 2	$O(n^2)$	$O(n^2)$	1:1
Level 3	$O(n^2)$	$O(n^3)$	1:n

複雑な線形計算を行うためのライブラリとして，LAPACK(Linear Algebra PACKage) がある．これらのライブラリは netlib[9] から入手可能であり，これ以外に各ベンダーは自社製品の性能を最大限に発揮できるよう最適化したライブラリをユーザに提供している．

図 2.5 にアーキテクチャの進歩とライブラリの動向を示す．LAPACK では LU 分解や QR 分解など，さまざまな行列分解を行うためのサブルーチンが含まれている．これらのルーチンはブロックアルゴリズムが用いられ，BLAS による行列・行列積を用いた高速な実装が行われている．ブロックアルゴリズムでは，行列をパネルと後続行列の 2 つに切り分け，

パネル部分で分解計算を行い，その結果を利用して後続行列部分の更新を行う．パネル分解と後続行列更新を繰り返し行うことで行列全体の行列分解を行う．後続行列の更新処理で並列実装された BLAS を用いて高速化を行うが，パネル分解の部分に逐次処理を含んでいる．つまりパネル分解計算がボトルネックとなる．第 1 章に述べたとおり，ブロックアルゴリズムは fork-join 型の並列計算モデルであるため並列計算資源のすべてを有効に活用することは原理的に不可能であり，近年の高並列なコンピュータのために別のパラダイムの普及が望まれている．

近年のマルチコアアーキテクチャに最適化された数値計算ライブラリの 1 つに PLASMA ライブラリ [3] がある．PLASMA ライブラリではタイルアルゴリズムによる並列化が行われている．タイルアルゴリズムは，細粒度のタスクを大量に生成し，これらを非同期に実行することですべての計算資源を休みなく動作させることを目的としており，近年の主流であるマルチコアアーキテクチャ向けアルゴリズムである．

クラスタシステムなどの分散メモリ型並列計算機向けの数値線形代数ライブラリとして ScaLAPACK(Scalable Linear Algebra PACKage)[10] が netlib から提供されている．BLAS，LAPACK と同様にベンダーから最適化された実装が提供されている．しかし，ScaLAPACK は 2019 年 11 月に提供されたバージョン 2.1.0 以降更新されていない．さらに，アルゴリズムは LAPACK と同様にブロックアルゴリズムを用いており，マルチコアアーキテクチャ向けに最適化されていない．クラスタシステム向けタイルアルゴリズムとして DPLASMA[11] が提供されている．しかし，2014 年以降の更新がなされていない．

ここまで述べた数値線形代数ライブラリは CPU を演算装置として用いたシステムのために実装されたライブラリである．近年の CPU/GPU ヘテロジニアス環境のための数値計算ライブラリとして，MAGMA (Matrix Algebra on GPU and Multicore Architectures) ライブラリ [12] が存在する．MAGMA では，LAPACK でも実装されているブロックアルゴリズムによる実装が行われている．ブロックアルゴリズムの主要演算である行列・行列積は GPU 向けの最適化が可能である．MAGMA ライブラリの行列分解では，ブロックアルゴリズムにおけるパネル分解計算を CPU で行い，後続行列更新は行列演算を高速に行うことが可能な GPU で計算することで高い性能を発揮させている．

さまざまな数値計算ライブラリがアーキテクチャの進化や，システム構成に向けて最適化されてきた．しかし，Summit など，近年のスーパーコンピュータの主流である CPU/GPU クラスタシステム向けの数値計算ライブラリは，現状では著者の知る限り存在しない．

表 2.3 にこれまで述べたライブラリを纏める．本研究では，CPU/GPU クラスタシステムでの高性能な線形計算ライブラリを構築することであり，行列分解の 1 つである QR 分解の高速な手法を行い，数値線形代数ライブラリの実装について考察を行う．

表 2.3: アーキテクチャと数値計算ライブラリ

	CPU	GPU	CPU/GPU
単一ワークステーション	PLASMA	cuSOLVER	MAGMA
クラスタシステム	DPLASMA/ParSEC ScaLAPACK		本研究

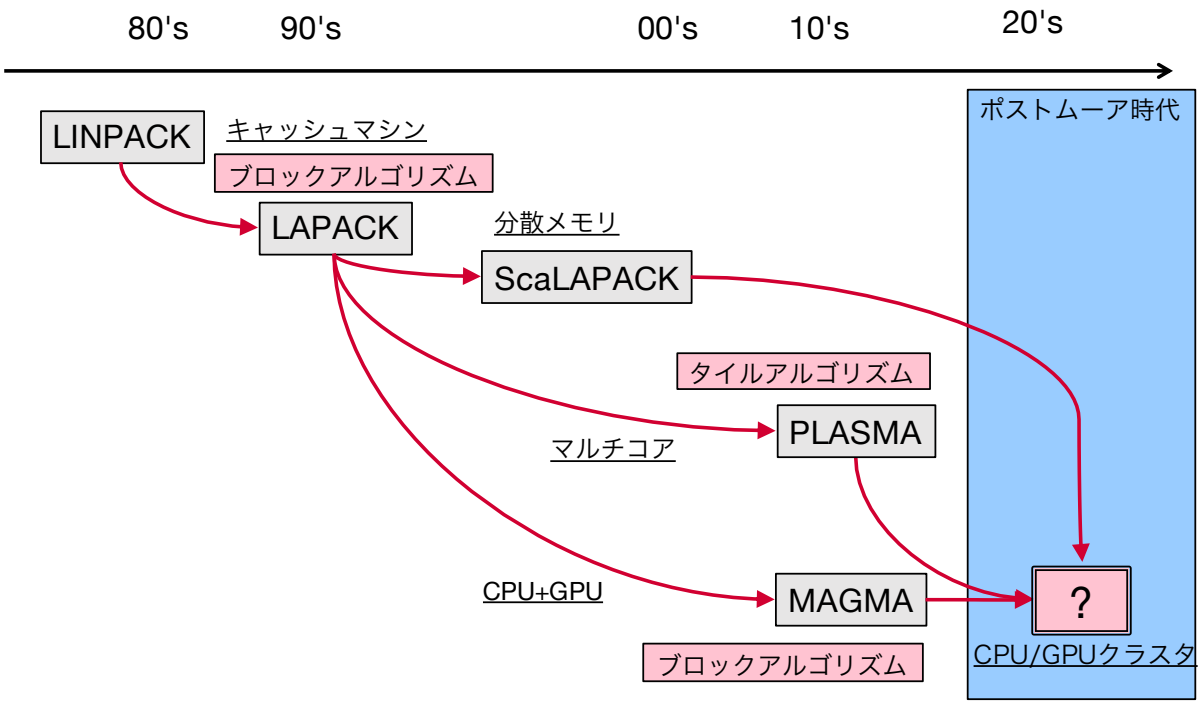

図 2.5: ハードウェアの進歩と数値計算ライブラリ

2.4 タイルアルゴリズムにおけるパラメータチューニング

前節でも述べたように，本研究ではクラスタシステムにおける数値線形代数ライブラリの並列化実装において，タイルアルゴリズムを用いる．タイルアルゴリズムにおいてタイルサイズと内部ブロック幅は非常に重要な性能パラメータである．タイルサイズは生成するタスクの量，内部ブロック幅は CPU のキャッシュ利用に影響を与える．問題サイズ，利用する計算環境により最適なタイルサイズと内部ブロック幅は異なる．最適なタイルサイズと内部ブロック幅を選択するためにはパラメータ探索を行う必要があり，パラメータ探索のために膨大な時間と電力消費が発生してしまう．そのため，短時間で最適なタイルサイズと内部ブロック幅の選択を行うことが求められている．

本研究の二つ目の目的は，マルチコアクラスタシステムにおいて，短時間でパラメータ探索可能な計算モデルを作成することである．

2.5 GPU アーキテクチャ

本研究では，CPU/GPU クラスタシステムを使用する．本研究で使用した 2 つの GPU アーキテクチャについてその特徴を以下に示す．

2.5.1 Pascal アーキテクチャ

Pascal アーキテクチャは 2016 年に発表された GPU アーキテクチャである．主な特徴として以下の点があげられる．

- 倍精度演算器 (FP64) の増加

- HBM2 メモリの使用
- NVLink による高速通信
- 反制度演算器 (FP16) 命令の追加

Pascal アーキテクチャである Tesla P100 は前世代の Maxwell の Tesla M40 と比較して，メモリ帯域が 3 倍の 720GB/sec，倍精度の性能は約 25 倍の 4.7TFLOPS の性能向上が得られている．また，NVLink によって GPU 間の接続を 40GB/sec で直接通信できる．東京工業大学学術国際情報センターの TSUBAME 3.0 や東京大学情報基盤センターの Reedbush-H などのクラスタシステムで使用されている．

2.5.2 Volta アーキテクチャ

Volta アーキテクチャは 2017 年に発表された GPU アーキテクチャである．主な特徴として以下の点があげられる．

- Tensor Core の追加
- FP32 命令と INT32 命令の同時実行

Volta アーキテクチャの Tesla V100 は前世代の Tesla P100 と比較して，Tensor Core の追加が大きな変更点である．Tensor Core では 4×4 の行列演算が可能となる．これにより，機械学習の分野では高速な演算が可能となっている．また倍精度演算器も増加しており，倍精度の性能は 7.5TFLOPS となり，Tesla P100 と比較して約 1.5 倍の性能向上がなされている．TOP500 ランキング 2 位の Summit や名古屋大学情報基盤センターの不老 Type II サブシステムなどに使用されている．

第3章　並列プログラミング

2章で述べたとおり，2000年代前半まで，CPUの性能はクロック周波数の向上によって得られており，プログラマは新しいコンピュータにプログラムを移行するだけで性能が得られる，いわゆるフリーランチの時代だった．2005年にIntel社は2つのPentium 4モジュールを一つのCPUとしてパッケージしたPentium Dプロセッサを発売した．電力の壁によりクロック周波数の上昇は停滞し，その後，複数のCPUコアを1つのダイに搭載するマルチコア化が進められており，近年までCPUの性能向上が行われてきた．CPUコア単体性能がほとんど変わらないため，現在では複数のCPUコアを効率的に使う並列プログラミング技術が必須となっている．この章では，並列プログラミング技術についてまとめる．

3.1　アーキテクチャの分類

1966年，フリンは命令とデータの並列度に基づいてコンピュータのアーキテクチャを分類した[13]．フリンの分類を下表に示す．

表 3.1: フリンのコンピュータアーキテクチャの分類

分類	概要
Single Instruction, Single Data stream (SISD)	逐次コンピュータ （命令，データに並列性なし）
Single Instruction, Multiple Data stream (SIMD)	1つの命令を複数のデータに適用する （ベクトルコンピュータ，GPU）
Multiple Instruction, Single Data stream (MISD)	1つのデータに複数の命令を適用する
Multiple Instruction, Multiple Data stream (MIMD)	複数の演算器が同時に複数の命令を複数のデータに適用する

近年では，複数のスレッドで1つの命令を実行するSingle Instruction, Multiple Thread (SIMT)や，Single Program, Multiple Data stream (SPMD)などの派生型が提案されている．

Intel社やAMD社のCPUでは，SSE拡張やAVX拡張と呼ばれるベクトル演算がサポートされている．ベクトル演算は複数のデータに対して同一の命令を適用するため，SIMD演算とも呼ばれており，SIMDは近年のコンピュータアーキテクチャにとって重要なコンセプトとなっている．データ並列性という概念で呼ばれることもある．

データ並列性と対比される形態にタスク並列性がある．これは命令もデータも異なる複数のタスクによる並列実行であり，フリンの分類ではMIMDに属する概念である．

3.1.1 並列化のレベル

表 3.2 に並列化のレベルを示す．

表 3.2: 並列化のレベル

1	マシンレベル
2	プロセスレベル
3	スレッドレベル
4	ベクトルレベル
5	命令レベル

並列プログラミングで扱うのはレベル 2 から 4 の一部であり，レベル 4 の一部から 5 の部分はコンパイラによって実行される．MPI による並列化はレベル 2，OpenMP による並列化はレベル 3 に相当する．MPI と OpenMP については後述する．

3.1.2 スレッド並列とプロセス並列

オペレーティングシステム上でアプリケーションプログラムを起動すると 1 つのプロセスが生成される．プロセスはそれぞれ固有のメモリ空間を持つ．スレッドはプロセスの中で並列に実行することが可能な命令の列である．プロセス内の複数のスレッドは，プロセスが持つメモリ空間を共有する．

複数のプロセスからなる並列プログラムをプロセス並列プログラムと言い，複数のスレッドで実行される並列プログラムをスレッド並列プログラムと言う．MPI では複数の MPI プロセスを生成し，マルチプロセス並列プログラムを作成する．スレッド並列プログラムを作成することで，マルチコア CPU 上で上記の並列化を行うことが可能となる．マルチスレッド並列プログラミングを行うために，かつては POSIX threads を用いて記述していた．POSIX threads ではスレッド生成，消滅をプログラム内に記述しなければならないことなどコード生成に関する制約が多く，プログラム作成の面で利用しやすいものではなかった．最近ではプログラム中の並列化可能な部分にディレクティブを挿入することで簡単に並列化可能な OpenMP が主流となっている．

3.1.3 メモリモデルによる分類

メモリモデルにより並列プログラミングを分類すると，共有メモリモデルと分散メモリモデルに分類できる．

共有メモリモデルでは，複数の演算器（またはプログラム）が同時にアクセス可能なメモリ（共有メモリ）を持つコンピュータを想定している．複数の演算器は同一のメモリにアクセス可能なため，演算器間の通信は不要となる．近年のマルチコア CPU は共有メモリ型並列計算機とみなすことが可能であり，OpenMP によるスレッド並列プログラムは共有メモリ型並列プログラミングモデルに分類される．Symmetric Multiprocessing (SMP) アーキテクチャは複数の CPU が同一のメモリ空間を共有し，どの CPU からも同じ時間でメモリにアクス可能である．SMP も共有メモリコンピュータに分類される．一方，複数の CPU がそれぞれ同一メモリ空間でアクセス可能な個別のメモリを持つ Nun Uniform Memory

Access (NUMA) アーキテクチャも共有メモリモデルに分類される．Nun Uniform とは，CPU（コア）によって高速にアクセス可能なメモリとそうでないメモリがあることを意味する．

分散メモリモデルでは，複数の演算器はそれぞれ個別のメモリを持つ．そのため，他の演算器が持つデータにアクセスするために通信が必要となる．MPI によるプロセス並列プログラムは分散メモリ型並列プログラミングモデルに分類される．

3.2 MPI

スーパーコンピュータに代表される現在の高並列計算環境の多くは分散メモリ型並列計算機に分類される．分散メモリ環境上の並列プログラムは複数のプロセスにより実行される．別のプロセス上にあるデータにアクセスするためには通信が必要となる．分散メモリ型並列計算機におけるデータ通信手段として，Message Passing Interface(MPI) が現在の主流である．MPI はメッセージ・パッシング用の規格の 1 つであり，主に複数の MPI プロセス間の通信に関する API を規定している．また，ほとんどの並列計算機では MPI がサポートされており，一度開発した MPI プログラムは複数の環境で無修正で実行可能なため可搬性が高いことも特徴である．MPI でプロセス並列を行い，OpenMP などでスレッド並列を行うことも可能であり，このようなプログラムは MPI/OpenMP ハイブリッド並列と呼ばれる．MPI は通信処理をユーザーが記述してアルゴリズムの最適化可能である．一方で，MPI によるデータ通信処理は演算速度と比較して非常に低速である．データ通信処理をいかに隠蔽するかが高速化の鍵となる．

3.3 OpenMP

OpenMP はマルチスレッドプログラミングの API であり，コンパイラが対応していればさまざまな環境でスレッド並列化を行うことができる．POSIX threads で必要なスレッドの生成や同期などの制御をユーザから隠蔽することができる．

Listing 3.1 に OpenMP によるループ並列化の例を挙げる．Listing 3.1 の 5 行目が OpenMP の並列化を行うディレクティブである．これにより，6 行目からの for 文は OpenMP で生成された複数スレッドによって並列に実行される．for 文の並列化はデータ並列の典型的な例であるが，本研究では OpenMP 4.0 以降で導入された task 構文と depend 節を用いる事でタスク並列プログラミングの作成を行った．

Listing 3.1: OpenMP によるスレッド並列化例

```
1  int main()
2  {
3    int a[100]
4
5    #pragma omp parallel for
6    for( int i=0; i<100; ++i)
7      a[i] = i;
8
9  }
```

3.4 task 構文

OpenMP 3.0 からタスク並列を行うための task 構文は導入されていたが，タスク間の制御を行うための機能が実装されていないため，あまり使い勝手の良い物では無かった．しかし，OpenMP 4.0 以降では depend 節が実装され，タスク間のデータ依存関係について指示ができるようになり，依存関係を考慮したタスク並列プログラミングが行えるようになった．依存関係を含めたタスクの記述方法は，タスクとして処理したいブロックの上に次の構文を記述する．

#pragma omp task depend (dependency-type: vars)

vars には依存する変数を指定する．dependency-type には次の 3 種類が指定できる．

- in
  指定された変数が入力データである時に指定

- out
  指定された変数が出力データである時に指定

- inout
  指定された変数が入出力データである時に指定

in は vars に指定した変数が生成するタスク以前の out,inout 指定されたタスクが終了していれば実行できる．out,inout は指定された変数が生成されるタスク以前の in,out,inout 指定されたタスクが終了していれば実行可能となる．簡単な動作例を Listing3.2 に載せる．例では 9 行目と 13 行目の演算が並列に実行される．17 行目の計算は，上 2 つの演算が終了した後に計算される．よって，各変数は x = 1，y = 3， z = 4 となる．しかし， depend 節を指定しなかった場合は 3 つのタスクは同時に実行されるため，意図した結果を得られず，各変数の値は予測できない．このように depend 節を追加することで，簡単に依存関係を考慮したタスク処理を行うことができる．

Listing 3.2: OpenMP task 節 depend 構文によるタスク並列例

```
1  int x=0,y=1,z=0;
2
3  #pragma omp parallel
4  {
5    #pragma omp master
6    {
7      #pragma omp task depend(out:x) depend(in:z)
8      {
9        x=z+1;
10     }
11     #pragma omp task depend(out:y) depend(in:z)
12     {
13       y=z+3;
14     }
15     #pragma omp task depend(in:x,y) depend(out:z)
16     {
17       z=x+y;
18     }
19   }
20 }
```

第4章　QR分解

$m \times n(m \geq n)$ の実数行列 A が与えられたとき，式4.1 のような分解が一意に存在する [14].

$$A = QR \tag{4.1}$$

ただし，Q は $m \times m$ 直交行列，R は $m \times n$ 上三角行列である．

QR 分解は最小二乗問題の安定な解法として利用されたり，特異値分解ための前処理として用いられる計算である．数値線形代数の重要な行列分解の 1 つであり，古くからさまざまな改良や応用が提案されている．QR 分解には次の三種類が存在する．

- ギブンス回転
- ハウスホルダー変換
- グラムシュミット直交化

ギブンス回転による直交変換は，対象となる行列の 1 つの要素のみ 0 にすることが出来る．行列の下三角部分を 0 化するギブンス回転を順に適用することにより，上三角行列 R を生成することが可能である．これはギブンス QR と呼ばれる．グラムシュミット直交化は一次独立な n 本の n 次元ベクトルから正規直交基底を生成するアルゴリズムである．ベクトルの直交性を利用して A の QR 分解を行うことが可能である．ハウスホルダー変換はこれらの中では比較的計算精度が高いことが知られている．本研究ではハウスホルダー変換による QR 分解を使用する．

4.1　ハウスホルダーQR分解

ハウスホルダー変換とは，式4.2 の H で定義される直交変換である．これを任意のベクトル a に作用させて，第一要素以外が 0 であるベクトル b を生成できる．

$$a = (a_0, a_1, \ldots, a_{n-1})^T$$

$$b = (b_0, 0, \ldots, 0)^T$$

$$v : \text{ベクトル}$$

$$t : \text{スカラ}$$

$$H : \text{ハウスホルダー行列}$$

$$I : \text{単位行列}$$

$$\|a\|_2 = \|b\|_2$$

$$v = a - b, t = \frac{2}{v^t v}$$

$$H = I - tvv^t$$

$$Ha = b \tag{4.2}$$

この変換手法を行列に繰り返し適用する事で，A を上三角行列 R に変形する行列分解がハウスホルダー QR 分解である．式 4.3 はハウスホルダー変換による QR 分解の 1 ステップを示している．行列 A の一番左側の列をベクトル a_0 として，a_0 の第一要素以外を 0 にするようなハウスホルダー行列 H_0 をもとめ，A の左側から適用する．最左列の先頭要素以外が 0 である行列 A' が求められる．同様の作業を A' の 1 列 1 行目以降の行列に対して繰り返し行うことで，上三角行列 R が求められる．直交行列 Q については，各列のベクトル a から生成された直交行列 $H_j, (j = 0, \ldots, n-2)$ の積から求まる (式 4.4)．LAPACK では上記作業を行うルーチンが提供されており，ハウスホルダーベクトルを求める関数は larfg，後続行列の更新は larf である．これらを利用した疑似コードを Algorithm 1 に示す．ハウスホルダー変換は，変換行列 $H = I - tvv^t$ を行列 A に左から適用するのではなく，$A - t(Av)v^t$ を実行する．これはランク 1 アップデートと呼ばれる演算で，行列・ベクトル積が主要な演算となる．ランク 1 アップデートは BLAS ライブラリで Level 2 に分類される演算である．Level 2 BLAS 演算は計算密度（＝演算回数/データ移動回数）が低いため CPU の演算性能を十分に発揮させることができない．そのため，多くの数値線形代数アルゴリズムでは計算密度の高い行列・行列積を利用するためにブロック化を用いた最適化を行っている．

$$H_0 A = H_0 \left(\begin{array}{c|ccc} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{array}\right) = A' = \left(\begin{array}{c|ccc} r_{0,0} & r_{0,1} & \cdots & r_{0,n-1} \\ \hline 0 & a'_{1,1} & \cdots & a'_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & a'_{m-1,1} & \cdots & a'_{m-1,n-1} \end{array}\right) \tag{4.3}$$

$$Q = H_0 H_1 \ldots H_{n-2} \tag{4.4}$$

Algorithm 1 ハウスホルダー QR 分解擬似コード

Require: $A \in \mathbb{R}^{m \times n} (m \geq n) A = (a_0, a_1, \ldots, a_{n-1})$

1: **for** $i = 0$ to $n - 1$ **do**
2: $\quad t, v \Leftarrow \mathrm{larfg}(a_i)$
3: $\quad A' \Leftarrow \mathrm{larf}(t, v, A)$
4: **end for**

4.2 ブロック QR 分解

ブロックアルゴリズム [15, 16] を用いたブロック QR 分解は図 4.1 のように行列 A を複数の列ごとに分割して扱う．この分割して切り出した部分行列をパネル，残りの行列を後続行列と呼ぶ．また，その列幅をブロックサイズと呼ぶ．ブロック QR 分解では，最初に切り出したパネルに対してハウスホルダー QR 分解を行う．ハウスホルダー QR 分解により，スカラー t，ベクトル v がブロックサイズ数分生成される．これらから，Algorithm2 のコンパクト WY 法 [17] により変換行列 T，V を生成する．この変換行列 T，V を用いた後続行列は，行列・行列積（gemm）と三角行列積（trsm）が主要演算となる．Level3 BLAS

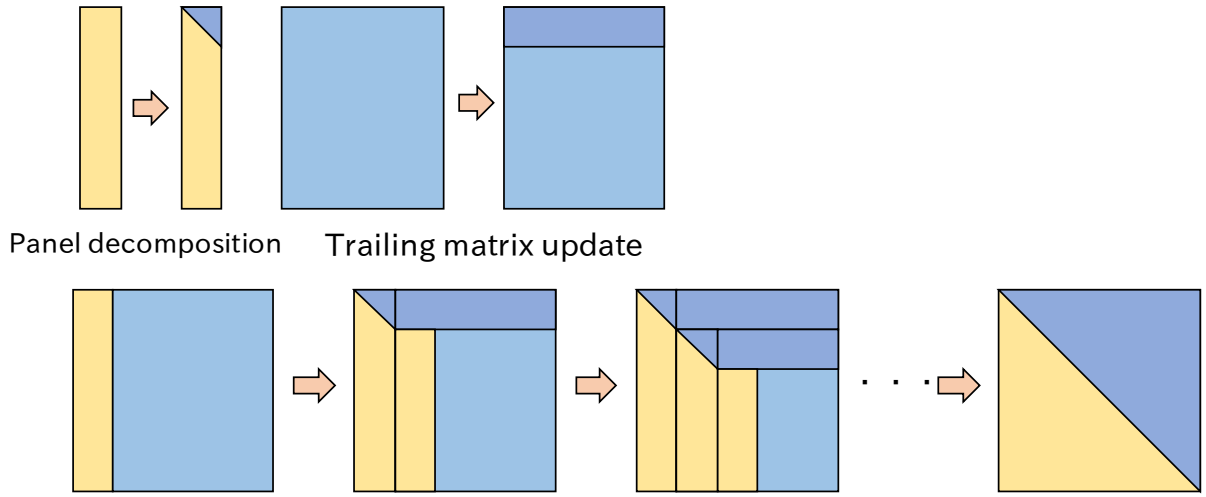

図 4.1: block algorithm の概要

演算である gemm と trsm は並列化実装された効率的なルーチンが提供されており，特に gemm の高速性により後続行列更新は逐次版と比較して高速となる．パネル分解，変換行列生成，後続行列更新を繰り返し行い，行列全体に対して行うことで QR 分解を行う．n これらのルーチンも LAPACK から使用することができ，パネル分解は dgeqr2，変換行列生成は larft，後続行列更新は larfb という関数名で提供されている．Algorithm 3 はこれらのルーチンを使用した擬似コードである．

ブロック QR 分解では，後続行列更新処理で並列化実装された行列・行列積を使用して計算効率を向上させたが，ハウスホルダー変換を用いたパネル分解はハウスホルダー QR で実行される．パネル分解の主要演算は行列ベクトル積であり，Level 2 BLAS 演算で実装される．ブロックアルゴリズムは，並列性の低いパネル分解と並列性の高い後続行列更新を順番に繰り返す fork-join 型並列プログラミングモデルに基づいている．このプログラミングモデルは，並列性の高い部分の演算で並列化されたルーチンを呼び出すのみで実現できるので，実装は容易である．しかし，並列性の低い部分があることから，現在主流のマルチコアアーキテクチャのような並列度の高いアーキテクチャの性能を十分に発揮させることができない．

Algorithm 2 Compact WY

Require: n is a block size

1: **for** $j = 0$ to $n - 1$ **do**
2: **if** $j = 1$ **then**
3: $V \Leftarrow [v_1]$
4: $T \Leftarrow [-t_1]$
5: **else**
6: $z \Leftarrow -t_j(T(V^T v_j))$
7: $V \Leftarrow (V v_j)$
8: $T \Leftarrow \begin{pmatrix} T & z \\ 0 & -t_j \end{pmatrix}$
9: **end if**
10: **end for**

Algorithm 3 ブロック QR 分解

Require: $A \in \mathbb{R}^{m \times n} (m \geq n)$

Require: s is a block size and q is number of panels. $q = n/s$

1: A is divided into n panels $A = [A_0 A_1, \ldots A_{q-1}]$
2: **for** $j = 0$ to $q - 1$ **do**
3: $\quad tau \Leftarrow geqr2(A_j)$
4: $\quad V_j, T_j \Leftarrow larft(A_j, tau)$
5: $\quad$ **if** $j \neq q - 1$ **then**
6: $\quad\quad [A'_{j+1} A'_{j+2} \ldots A'_{q-1}] \Leftarrow larfb([A_{j+1} A_{j+2} \ldots A_{q-1}], V_j, T_j)$
7: $\quad$ **end if**
8: **end for**

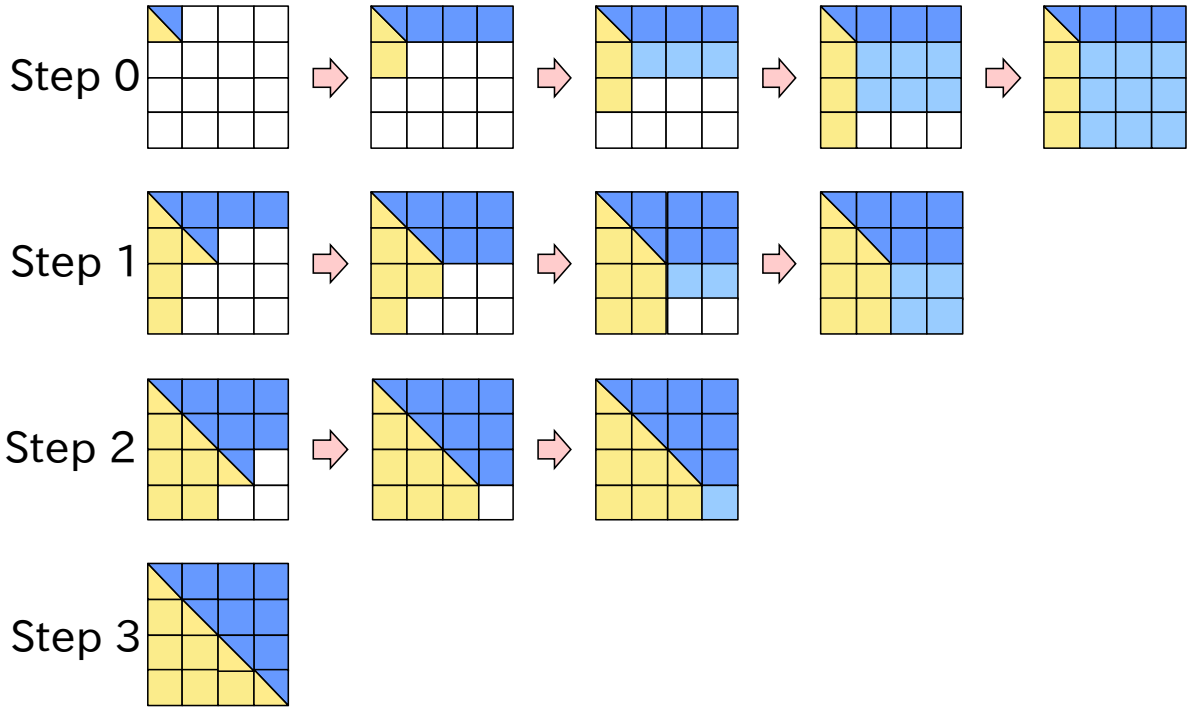

図 4.2: tile algorithm の概要

4.3 タイル QR 分解

タイルアルゴリズム [18, 19, 20] は，ブロックアルゴリズムの並列度をさらに高めたアルゴリズムである．タイルアルゴリズムでは，行列を行方向と列方向に分割を行い，図 4.2 のようにタイルと呼ばれる小行列を作成する．各タイルごとに計算カーネルと呼ばれる処理を行うことで QR 分解を行う．計算カーネルにはデータ依存が存在する．これについては後述する．データ依存が解消されたカーネルは任意の順序で計算を行うことができるという特徴がある．本研究ではタイルアルゴリズムを用いた QR 分解で並列度を高める．

以下では行列 A のサイズを $pb \times qb$ と定義し，行列 A を次のように表す．

$$\begin{pmatrix} A_{0,0} & A_{0,1} & \cdots & A_{0,q-1} \\ A_{1,0} & A_{1,1} & \cdots & A_{1,q-1} \\ \vdots & \vdots & \ddots & \vdots \\ A_{p-1,0} & A_{p-1,1} & \cdots & A_{p-1,q-1} \end{pmatrix} \tag{4.5}$$

ここで，b はタイルサイズ，各タイル $A_{i,j}$ は $b \times b$ の正方行列である．タイル QR 分解は Algorithm 4 の擬似コードで実装できる．

Algorithm 4 タイル QR 分解擬似コード

Require: $A \in \mathbb{R}^{pb \times qb} (p \geq q)$
Require: b is a tile size and p, q are the number of tiles columns or rows.
1: **for** $k = 0$ to $min(p, q)$ **do**
2: GEQRT($A_{k,k}$)
3: **for** $j = k + 1$ to $q - 1$ **do**
4: LARFB($A_{k,j}$)
5: **end for**
6: **for** $i = k + 1$ to $p - 1$ **do**
7: TSQRT($A_{k,k}, A_{i,k}$)
8: **for** $j = k + 1$ to $q - 1$ **do**
9: SSRFB($A_{k,j}$)
10: **end for**
11: **end for**
12: **end for**

以下では，タイル QR 分解を構成する 2 種類 4 つの計算カーネルについて説明する．説明内に出てくるタイルのインデックスはアルゴリズム 4 と対応している．

4.3.1 GEQRT カーネル

対角タイルに QR 分解を適用するカーネルである．図 4.2 の対角タイル $A_{k,k}$ にハウスホルダー変換を用いた QR 分解を行い，上三角行列 $R_{k,k}$ とコンパクト WY 法による変換行列 $V_{k,k}$ 行列，$T_{k,k}$ 行列を生成する．$R_{k,k}$ は $A_{k,k}$ に上書きされ，単位下三角行列 $V_{k,k}$ は，その非対角要素を要素が 0 となった $A_{k,k}$ の下三角部分に保存し，$T_{k,k}$ は別に確保した領域に保存する．

$$(R_{k,k}, V_{k,k}, T_{k,k}) \leftarrow \text{GEQRT}(A_{k,k})$$
$$A_{k,k} \leftarrow (R_{k,k}, V_{k,k}),\ T_{k,k} \leftarrow T_{k,k}$$

4.3.2 TSQRT カーネル

GEQRT を適用した上三角タイル $A_{k,k}$ と，同じタイル列の下のタイル $A_{i,k}, (i > k)$ を組み合わせて QR 分解を実行するカーネルである．GEQRT と同様に，変換行列として $V_{i,k}$ 行列と $T_{i,k}$ 行列を生成する．正方行列 $V_{i,k}$ は要素が 0 となった $A_{i,k}$ 部分に格納され，$T_{i,k}$ 行列は別途確保した領域に保存する．

$$(R_{k,k}, V_{i,k}, T_{i,k}) \leftarrow \text{TSQRT}(R_{k,k}, A_{i,k})$$
$$R_{k,k} \leftarrow R_{k,k},\ A_{i,k} \leftarrow V_{i,k},\ T_{k,k} \leftarrow T_{k,k}$$

4.3.3 LARFB カーネル

GEQRT カーネルを適用したタイル列の右側のタイル $A_{k,j}, (j > k)$ に適用するカーネルである．GEQRT カーネルで生成された変換行列 $V_{k,k}$ 行列，$T_{k,k}$ 行列を使用して，式 4.6 による更新を行う．

$$A_{k,j} \leftarrow \left(I - V_{k,k} T_{k,k} V_{k,k}^T\right) A_{k,j} \tag{4.6}$$

4.3.4 SSRFB カーネル

LARFB カーネルを適用したタイル $A_{k,j}$ と，TSQRT を適用したタイル列の右側タイル $A_{i,j}, (i, j > k)$ に適用するカーネル．TSQRT カーネルで生成された $V_{i,k}$ 行列，$T_{i,k}$ 行列を用いて，式 4.7 による更新を行う．

$$\begin{pmatrix} A_{k,j} \\ A_{i,j} \end{pmatrix} \leftarrow \left(I - \begin{pmatrix} I \\ V_{i,k} \end{pmatrix} T_{i,k} \begin{pmatrix} I & V_{i,k}^T \end{pmatrix} \right) \begin{pmatrix} A_{k,j} \\ A_{i,j} \end{pmatrix} \tag{4.7}$$

これら 4 つのカーネルのうち，QR 分解を行う GEQRT，TSQRT カーネルを分解カーネル，変換行列を用いてタイル更新を行う LARFB，SSRFB カーネルを更新カーネルと呼ぶ．また，これらのカーネルの内部ではシングルスレッド版の BLAS ルーチンを使用する．

各カーネルはブロックアルゴリズムで実装される．このブロック幅を内部ブロック幅と呼ぶ．ブロックアルゴリズムにおいてブロック幅が重要な性能パラメータであることから，タイルアルゴリズムの内部ブロック幅も重要な性能パラメータとなる．

各カーネルの計算量 [21] と呼び出し回数を表 4.1 に示す．ただし，$p = q$ とした．

表 4.1: タイル QR 分解カーネルの計算量と呼び出し回数

カーネル	計算量	呼び出し回数
GEQRT	$2b^3$	p
TSQRT	$10b^3/3$	$p(p-1)/2$
LARFB	$3b^3$	$p(p-1)/2$
SSRFB	$4b^3$	$p(2p-1)(p-1)/6$

表 4.1 より，タイル QR 分解における主要演算は SSRFB カーネルによる後続行列更新であることが分かる．

4.3.5 依存関係

各カーネルにはデータ依存があり，依存関係が解消されたカーネルは実行可能である．カーネルの依存関係には 3 種類存在する．図 4.2 において，縦方向の依存関係を i 方向依存，横方向の依存関係を j 方向依存，同一タイルに異なるカーネルを適用する依存関係を k 方向依存と呼ぶ．それぞれの方向は Algorithm 4 のループインデックスと対応している．3 種類のデータ依存は以下の通り．

- i 方向依存
  同一タイル列に対するカーネル実行では，最上タイルが常に更新される．したがって，同一タイル列に対するカーネルは逐次実行される．

- j 方向依存
  LARFB，SSRFB カーネルは同一タイル行の GEQRT，TSQRT カーネルが生成する変換行列を使用するので，これら分解カーネルの処理の後でなければ実行できない．ただし，同一タイル行に対する更新カーネルは並列実行可能．

- k 方向依存
  同一タイルに対する k+1 ステップのカーネルは，k ステップ目のカーネルが終了した後でなければ実行できない．

4.4 動的スケジューリング

各計算カーネルは依存関係が解消されなければ実行できない．ループなどの制御文によらず，依存関係を監視することで実行可能となったタスクから処理を行うスケジューリングを動的スケジューリングと呼ぶ．古いバージョンの PLASMA ライブラリでは，QUARK[22] と呼ばれるタスクスケジューラによりタイルアルゴリズムにタスクの動的スケジューリングを導入していた．また，依存関係の状況が記されたプログレステーブルと，実行可能となったカーネルの位置情報を保存するスケジュールキューを用いることでも動的スケジューリングの実装が可能である [23]．しかし，これらの実装は依存関係の監視と更新をプログラム内に記述する必要があるため，ソースコードが煩雑になりプログラムの生産性が悪い．現在，動的タスクスケジューリングは，OpenMP 4.0 から導入された task 構文と depend 節を逐次コードに導入することで実装可能である．task 構文で指定されたブロックがタスクとして実行され，depend 節で表されたデータ依存を監視して，依存関係が解決されたタスクから実行される．

Algorithm4 を j-loop 並列化したコード例と task depend による並列化コード例を Listing4.1,4.2 に示す．depend 節に指定する変数は，Array Sections 表現により配列領域を指定する事ができる．

Listing 4.1: Algorithm4 の OpenMP による fork-join 型並列化

```
1  for(int k=0; k<min(p,q); ++k){
2    #pragma omp parallel
3    {
4      #pragma omp single
5      {
6        GEQRT(A(k,k));
7      }
8      #pragma omp for
9      for( int j=k+1; j<q; ++j)
10       LARFB(A(k,j),V(k,k),T(k,k));
11
12     for( int i=k+1; i<p; ++i){
13       #pragma omp single
14       {
15         TSQRT(A(k,k),A(i,k))
16       }
17       #pragma omp for
18       for( int j=k+1; j<q; ++j)
19         SSRFB(A(k,j),A(i,j),V(i,k),T(i,k));
20     }
21   }
22 }
```

Listing 4.2: Algorithm4 の OpenMP task 節 depend 構文によるタスク並列化

```
1  #pragma omp parallel
2  {
3    #pragma omp master
4    {
5      for(int k=0; k<min(p,q); ++k){
6        #pragma omp task depend(inout:A(k,k)) depend(out:V(k,k),T(k,k))
7        GEQRT(A(k,k));
8
9        for( int j=k+1; j<q; ++j){
10         #pragma omp task depend(in:V(k,k),T(k,k)) depend(inout:A(k,j))
11         LARFB(A(k,j),V(k,k),T(k,k));
12       }
13
14       for( int i=k+1; i<p; ++i){
15         #pragma omp task depend(inout:A(k,k),A(i,k)) depend(out:V(i,k),T(i,k))
16         TSQRT(A(k,k),A(i,k))
17
18         for( int j=k+1; j<q; ++j){
19           #pragma omp task depend(in:V(i,k),T(i,k)) depend(inout:A(k,j),A(i,j))
20           SSRFB(A(k,j),A(i,j),V(i,k),T(i,k));
21         }
22       }
23     }
24   }
25 }
```

図 4.3，4.4 に Listing4.1 と 4.2 の性能を示す．使用した CPU は Intel Core i7-6900K（Broadwell E, 3.2GHz, 8 core, 8 スレッド実行）である．タイル QR のカーネルは PLASMA 17.1 のものを使用し，BLAS ライブラリは Intel MKL 2018.0.128 を使用した．同図において，横軸は行列サイズ，縦軸は速度を表す．タイルサイズは 80, 160, 320, 480, 640 としている．

これより，動的スケジューリングによりタイル QR の性能が 20%程度上昇していることが分かる．また，タイルサイズが性能に及ぼす影響も見ることができる．j ループ並列ではタイルサイズ 80 と 320 の速度差は最大 45%，動的スケジューリングではタイルサイズ 80 と 320 で 63%の最大性能差がある．

j ループ並列では行列サイズが小さいときに性能が低い．これはタスク数が少ないため，休止状態となっている CPU コアが多いためだと考えられる．これに対して動的スケジューリングでは，データ依存を監視することで実行可能なタスクを抽出することができるため，行列サイズが小さいときから最大性能に近い速度が出ている．

タイル QR の 4 つのカーネルのうち最も呼び出し回数が多いのは SSRFB カーネルであり，SSRFB カーネルの主要演算は行列・行列積演算である．BLAS ライブラリの行列・行列積演算は比較的大きなサイズ向けに最適化されているため，タイルサイズは大きいほど SSRFB カーネルは高速に実行される．一方，タイルサイズが小さいほどタスク数は多くなり，計算資源への負荷分散は良好となる．図 4.3 の，タイルサイズが小さい方が高速であるという行列サイズが小さいときの挙動はこのように説明できる．

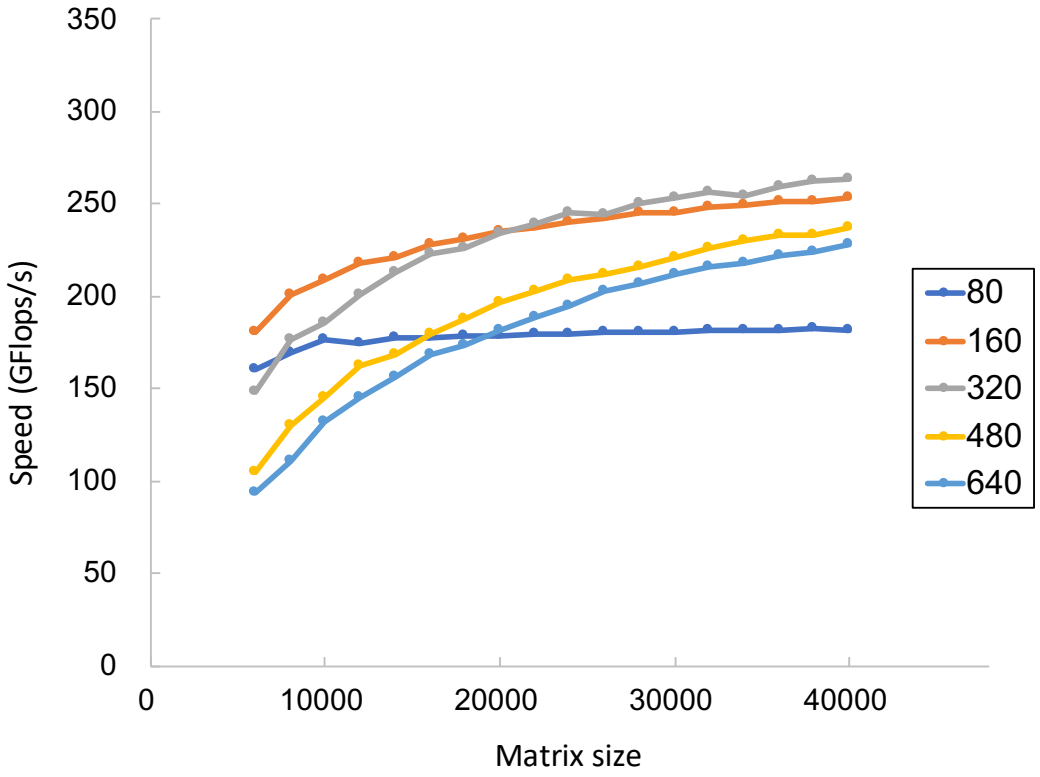

図 4.3: j ループ並列の速度

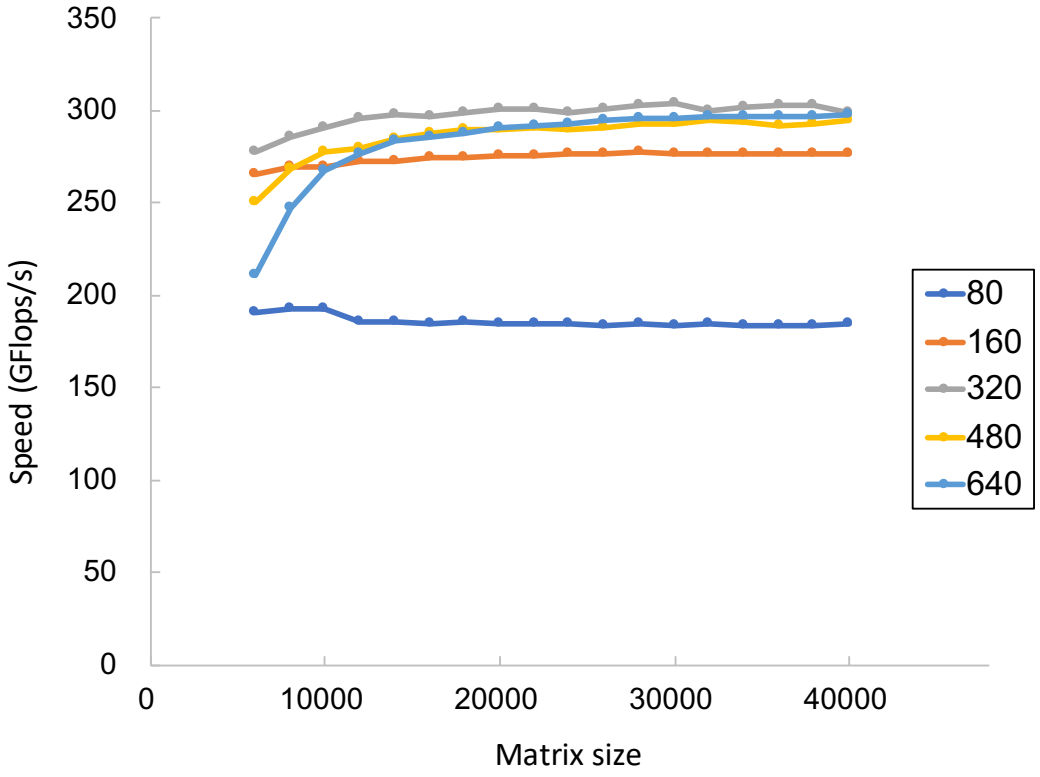

図 4.4: 動的スケジューリングの速度

第5章　高並列環境での実装

5.1　Communication-Avoiding QR

これまでに，タイルアルゴリズムはマルチコア CPU が搭載された単体のワークステーションのような共有メモリ環境上では高い性能が得られていることを示した．一方，通信ネットワークで接続された複数ノードからなる並列コンピュータのような分散メモリ環境上では，i 方向の逐次依存性やデータ通信処理がボトルネックとなり性能が伸びない．本研究においてもマルチコアクラスタ，CPU/GPU ヘテロジニアスクラスタシステム上でハイブリッド MPI 方式の並列プログラミングを行った．

一般的に，演算速度と比較して，クラスタシステムのノード間通信速度は非常に低速である．タイル QR 分解では，TSQRT や LARFB，SSRFB カーネルはカーネル実行時に変換行列や各ステップの最上位タイルを必要とする．タイル QR 分解では，分解カーネルを実行するたびにこのようなデータ通信が発生するため，大規模並列環境では性能を発揮しきれない．この問題に対して，通信量削減手法 (Communication Avoiding) をタイル QR 分解に用いる．

Communication-Avoiding QR 分解 (CAQR)[24, 25, 26, 27] では図 5.1 の様にタイル行列を i 方向に複数の領域に分割する．この領域をドメインと呼ぶ．それぞれのドメインでタイル QR 分解の 1 ステップを行い，各ドメインの最上位タイル行の上三角化を行う．その後，各ドメインにの最上位タイル行に対してタイル QR 分解を行い，上三角化を行う．CAQR を行うことでタイル QR 分解よりも計算処理が増えるが，プロセス間の通信回数が削減できるため分散メモリ型並列計算機では性能が発揮できる．

CAQR では，QR 分解の 4 カーネルに加え，ドメイン間のマージを行う 2 カーネルが追加される．

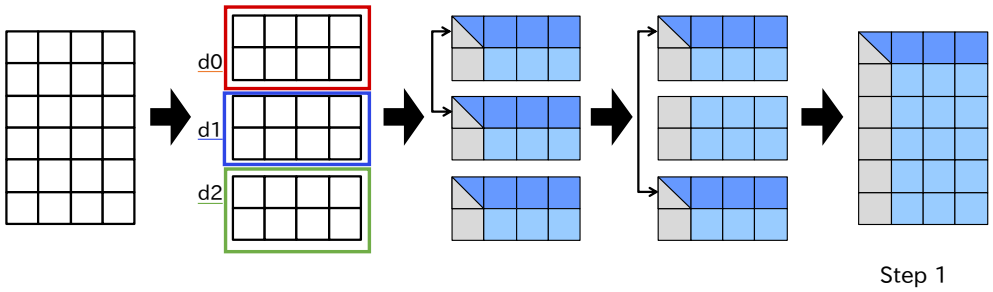

図 5.1: CAQR アルゴリズムの概要．3 ドメインに分割を行った場合の 1 ステップ内の実行順序．

5.1.1 TTQRT カーネル

GEQRT を適用した上三角タイル $R_{i0,k}$ と $R_{i1,k}$ を組み合わせて QR 分解を実行するカーネル．変換行列 $V_{i1,k}$ 行列と $T_{i1,k}$ 行列を生成する．$V_{i1,k}$ は要素が 0 となった $R_{i1,k}$ 部分に転置して格納され，$T_{i1,k}$ 行列は別途確保した領域に保存する．

$$(R_{i0,k}, V_{i1,k}, T_{i1,k}) \leftarrow \mathrm{TTQRT}(R_{i0,k}, R_{i1,k})$$
$$R_{i0,k} \leftarrow R_{i0,k} R_{i1,k} \leftarrow V_{i1,k}^T T_{i1,k} \leftarrow T_{i1,k}$$

5.1.2 TTMQR カーネル

LARFB カーネルを適用したタイル $A_{i0,j}$ と，$A_{i1,j}$ に適用するカーネル．TTQRT カーネルで生成された $V_{i1,k}$ 行列，$T_{i1,k}$ 行列を用いて，式 5.1 による更新を行う．

$$\begin{pmatrix} A_{i0,j} \\ A_{i1,j} \end{pmatrix} \leftarrow \left(I - \begin{pmatrix} I \\ V_{i1,k} \end{pmatrix} T_{i1,k} \left(I V_{i1,k}^T \right) \right) \begin{pmatrix} A_{i0,j} \\ A_{i1,j} \end{pmatrix} \tag{5.1}$$

第6章　タイルサイズチューニング

タイルアルゴリズムにおいて，タイルサイズは非常に重要なパラメータであり，計算速度に多大な影響を与える．しかし，大規模な並列環境下で最適なタイルサイズを得るためのパラメータ探索を行うことは，非常にコストがかかる．ここでは，マルチコアクラスタシステムにおけるパラメータチューニングを行うことを目的として，最適なタイルサイズ探索のためのタイルCAQRの性能モデルを構築した．本課題については，理化学研究所計算科学研究センター(R-CCS)が所有する京コンピュータで行った．[23, 28] CAQRアルゴリズム全体の実行時間T_{all}は次の実行時間の総和である．

- ドメイン内タイルQR分解の計算時間T_{QR}
- ドメイン間マージ処理の計算時間T_{merge}
- ドメイン間の通信時間T_{comm}

各実行時間に対するモデルを立て，全体の性能モデルを構築する．次のような状況を仮定する．

- 行列サイズ$m \times n$
- タイルサイズ$b \times b$
- タイル数$p \times q(p = \lceil m/b \rceil q = \lceil m/b \rceil)$
- ノード数P
- ノードあたりのCPUコア数C
- ドメイン内タイル行数$d(d = p/P, d \geq q, pmodP = 0)$

6.1　ドメイン内タイルQR分解の計算時間

ドメイン内タイルQR分解は動的スケジューリングによって実装されており，非同期に複数のタスクがオーバーラップして実行されるため正確な性能モデルの構築を行うことは難しい．タイルQR分解においてもっとも支配的なカーネルはSSRFBカーネルである．例えば，4000×1000の縦長行列に対して，タイルサイズ100×100におけるタイルQR分解のカーネルの割合を調べると，SSRFBカーネルの比率は約80%となる．そこで，今回はSSRFBカーネルの実行回数から，ドメイン内タイルQR分解の性能モデルを構築する．第kステップにおけるSSRFBの実行回数は，全CPUコアを使用して第i行のSSRBカーネル全て実行するのにかかる回数，$\lceil (q-k-1)/C \rceil$と第kステップにおけるタイル行数

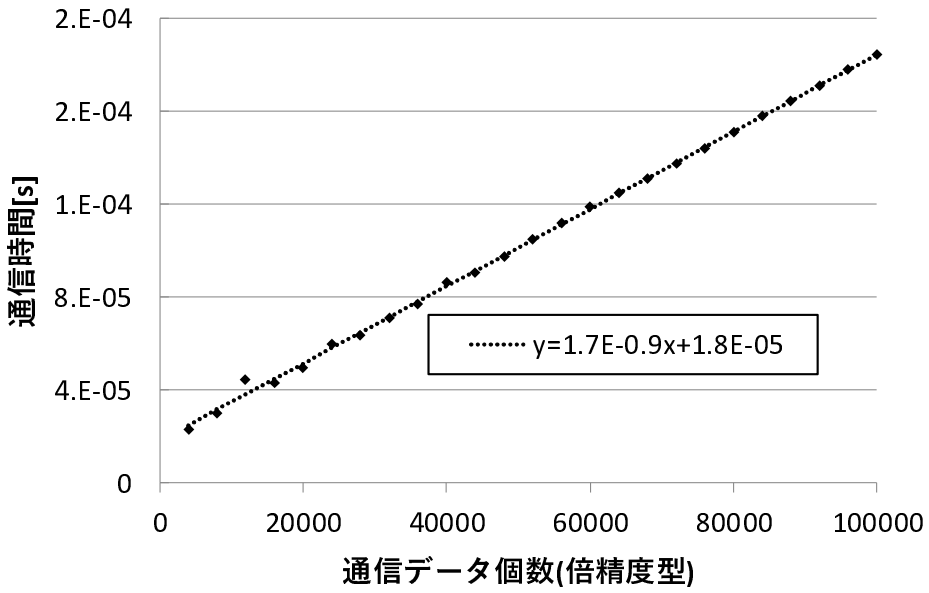

図 6.1: 2 ノード間の MPI_Send の実行時間

$d - \lfloor k/P \rfloor$ の積である．よって，タイルサイズを $b \times b$ のときの SSRFB カーネル実行時間を T_{SSRFB} としたとき，ドメイン内タイル QR 分解の実行時間の性能モデルは次の式となる．

$$T_{QR} = T_{SSRFB} \times \sum_{k=0}^{q-1} (d - \lfloor k/P \rfloor) \times \lceil (q-k-1)/C \rceil \tag{6.1}$$

6.2 ドメイン間マージ処理の計算時間および通信時間

ドメイン間マージ実行時間は計算時間 T_{merge} とドメイン間通信時間 T_{comm} の和である [29, 30]．第 k ステップにおける 1 回のマージ処理にかかる時間は，TTQRT の実行時間を T_{TTQRT}，TTMQR を T_{TTMQR} とすると，$T_{TTQRT} + \lceil (q-k-1)/C \rceil \times T_{TTMQR}$ となる．また，ドメイン間の通信時間については alpha-beta model を用いる．これは，通信の遅延を α，ネットワークのバンド幅の逆数を β，通信するデータサイズを n としたとき，通信時間を $\alpha + \beta n$ とモデル化するものである．図 6.1 は京コンピュータ 2 ノード間における MPI の一対一同期通信 MPI_Send について double 型のデータ数を変化させ通信時間を測定したグラフである．この図には最小二乗法で得られた回帰直線も併記させており，その結果から，通信モデルのパラメータは $\alpha = 1.8 \times 10^{-5}$ 秒，$\beta = 1.7 \times 10^{-9}$(秒/個) である．$\alpha, \beta$ の値から，通信一回を行うのに必要な立ち上がり時間 α は，データ一個あたりの送信コスト β よりも非常に大きいことが確認できる．この通信モデルから，第 k ステップにおけるマージ処理一回行うときにかかる送受信の通信時間は $2 \times (\alpha + \beta b^2 (q-k))$ となる．マージ処理におけるリダクションツリーのマージ回数を N とすると，実行時間は次の通りである．

$$T_{merge} = \sum_{k=0}^{q-1} (T_{TTQRT} + T_{TTMQR} \times \lceil (q-k-1)/C \rceil) N \tag{6.2}$$

$$T_{comm} = \sum_{k=0}^{q-1} 2(\alpha + \beta b^2 (q-k)) N \tag{6.3}$$

式 6.1 から 6.3 の総和が計算モデルとなる [23, 31, 32]．本実験の結果は 8.1 に示す．

第7章　GPU利用

近年では，GPU を追加の演算装置として利用するのが定着している．機械学習，特に Deep Learning の分野では GPU を用いて学習を行う事が一般的である．科学技術計算の分野では，いち早く GPU の持つ演算能力に着目し，2000 年代後半から GPGPU として利用し始めた．GPU の演算能力の特徴として，計算コアの 1 つあたり性能は低いが，大量の計算コアを有している．そのため，行列行列積などのデータ並列なタスクに対して非常に高い性能を発揮することができる．CPU/GPU ヘテロジニアス環境においてタイルアルゴリズムを用いた高速化手法も行われてきた [33, 34]．この章では，GPU の特徴からタイルアルゴリズムにおける GPU 利用の手法について述べる．

7.1　MAGMA ライブラリの実装

先行研究である MAGMA ライブラリ [12] の QR 分解では，ブロックアルゴリズムを用いている．MAGMA ではパネル分解処理を CPU に，後続行列更新処理を GPU に割り当てている．ブロックアルゴリズムによる後続行列更新は，タイルアルゴリズムによる更新処理と比較して，比較的大きな行列に対して更新処理を行う．そのため，GPU を利用することで高い性能が得られる．また，パネル分解についてはマルチスレッド版 BLAS を用いている．

7.2　更新カーネルの最適化

後続行列の更新を行うカーネルについては既に前章で述べた．式 7.1 から 7.4 は更新カーネルの 1 つ SSRFB カーネルを BLAS ルーチンを用いて実装したものである．ただし，W は作業用領域である．

$$W = A_{k,j} + V_{i,k}^T A_{i,j} \qquad (gemm) \tag{7.1}$$

$$W = T_{i,k}^T W \qquad (trmm) \tag{7.2}$$

$$A_{k,j} = A_{k,j} - W \qquad (axpy) \tag{7.3}$$

$$A_{i,j} = A_{i,j} - V_{i,k} W \qquad (gemm) \tag{7.4}$$

このように，更新カーネル SSRFB の主要演算は Level3 BLAS の gemm と trmm である．GPU は並列処理の性能が高いので， Level3 BLAS を多く使用しており，並列に実行が可能な更新カーネルを GPU に割り当て，逐次処理が多い分解カーネル及び GPU の制御を CPU に割り当てる．Listing 7.1 は PLASMA カーネルを参考にして作成した SSRFB カーネルである．ここでは簡略化のために内部ブロック化は省略している．GPU では，特に Level1 BLAS の処理 copy，axpy が非常に遅い．そこで，Listing 7.2 のように cuBLAS で実装されている BLAS の拡張カーネルである geam に書き換えることで最適化を行った

[35]．他の更新カーネル LARFB，TTMQR カーネルに関しても同様に geam への書き換えを行った．図 7.1 はこの後に述べる Bulk Update 手法を用いた Reedbush-H 1 ノードにおけるタイル CAQR アルゴリズムのカーネル最適化前後の性能評価結果である．MPI プロセスを 2 プロセス立ち上げ，性能評価を行った．この最適化によりタイル CAQR アルゴリズムの性能はピーク時で約 20%向上した．

Listing 7.1: PLASMA カーネルを元にした SSRFB 実装

```
1  int SSRFB(double *V, double *T, double *Akj, double *Aij, double *WORK){
2    //WORK = Akj
3    cublasDcopy( Akj, WORK );
4
5    //WORK = V**T * Aij + WORK
6    cublasDgemm( Aik**T, Aij, WORK );
7
8    //WORK = T**T * WORK
9    cublasDtrmm( T**T, WORK );
10
11   //Akj = Akj - WORK
12   //タイルサイズ b
13   for( int i=0; i<b; ++i)
14     cublasDaxpy( Akj[i], WORK[i] )
15
16   //Aij = Aij - V * WORK
17   cublasDgemm( V, WORK, Aij );
18 }
```

Listing 7.2: GPU 向け最適化を施した SSRFB 実装

```
1  int SSRFB(double *V, double *T, double *Akj, double *Aij, double *WORK){
2    //WORK = Akj
3    cublasDgeam( Akj, WORK );
4
5    //WORK = V**T * Aij + WORK
6    cublasDgemm( Aik**T, Aij, WORK );
7
8    //WORK = T**T * WORK
9    cublasDtrmm( T**T, WORK );
10
11   //Akj = Akj - WORK
12   cublasDgeam( Akj, WORK )
13
14   //Aij = Aij - V * WORK
15   cublasDgemm( V, WORK, Aij );
16 }
```

また，本研究では大規模な密行列に対して行列分解計算を行うために，更新カーネルが必要とする行列データのみを GPU メモリに配置する．そのため，GPU を使用した更新処理のためには CPU-GPU 間で変換行列および更新行列のデータ移動が，更新行列実行のたびに必要となる．一方，MAGMA ライブラリでは基本的にすべての行列データを GPU メモリ上に配置する．(GPU メモリサイズを越える大きさの行列を扱うルーチンも存在する)

更新カーネルの GPU 実装に関して 2 種類の実装を行った．

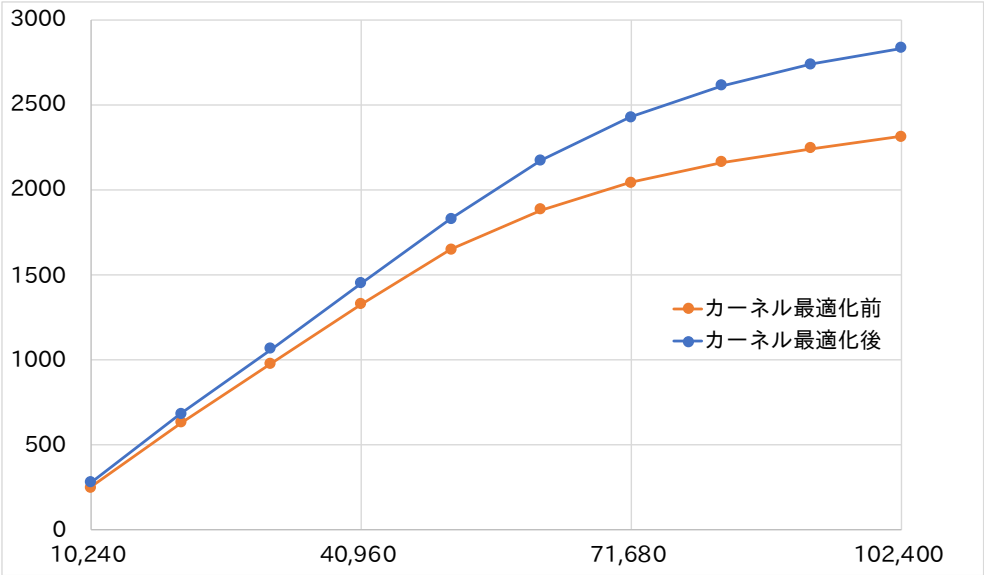

図 7.1: dgeam ルーチン使用前後における正方行列に対するタイル CAQR アルゴリズムの性能評価．横軸は行列サイズ，縦軸は計算速度．

7.3 Bulk Update

これまでの GPU プログラミングでは 1 つの計算ルーチンを GPU 全スレッドを使用して並列計算を行うことで，GPU の計算性能を生かしてきた．そのため，GPU のすべての計算資源が稼働状態となるような大きな行列に対して行列演算を行うことでより高い処理性能が得られる．しかし，タイル CAQR アルゴリズムにおいて分解カーネルを CPU，更新カーネルを GPU で処理する場合，GPU の計算能力を発揮させるためにタイルサイズを大きくすると，分解カーネルの処理に時間がかかり GPU での処理の開始が遅延してしまう．そこで，小さいタイルサイズを選択しつつ GPU の性能処理を発揮する方法を提案した [36, 37]．これを Bulk Update と呼ぶ．Bulk Update では図 7.2 のように，並列実行可能な j 方向のタイルを 1 つの長方行列にまとめて，一度に更新カーネルを適用する．長方行列にまとめて更新を行うことで， GPU での更新カーネルの性能を向上できると考えられる．最上タイル行は，LARFB 実行後も SSRFB カーネルで更新を行うため，k ステップの全 SSRFB 実行が終わるまで GPU メモリに残しておく．また，GPU メモリには 2 タイル行分の，変換行列および作業用行列のデータのみ配置を行うため，比較的容量の少ない GPU メモリでも大規模な行列を扱う事が可能である．また，更新カーネル適用データについては double buffering を行う事で，データ通信の隠蔽を行う事も可能である．

7.4 Stream Update

NVIDIA 社の Kepler アーキテクチャ以降の GPU では HyperQ により，複数タスクを同時実行できるようになっている．異なる CUDA ストリームに複数の GPU タスクが投入された時，実行可能ならばそれらが自動的に並列実行される．そこで図 7.3 のように，各タイル列毎に CUDA ストリームを作成し，それぞれのストリームに各タイル列の更新タスクを割り当てる．異なる CUDA ストリームに割り当てることで並列実行可能なタスクが非同期に更新カーネルを実行することが可能となる．Bulk Update では更新タスクの粒度を大きくしてしまうため，タイルアルゴリズムの「細粒度タスクの非同期実行」という特徴と相

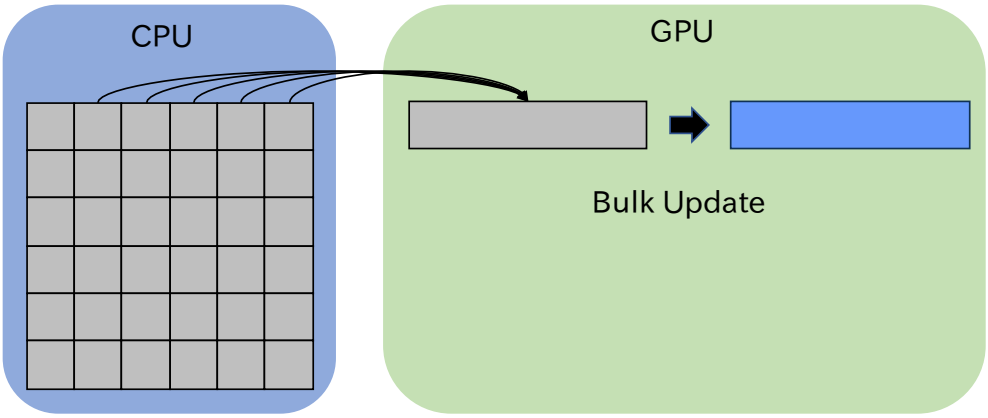

図 7.2: Bulk update 1 タイル列の更新処理手法

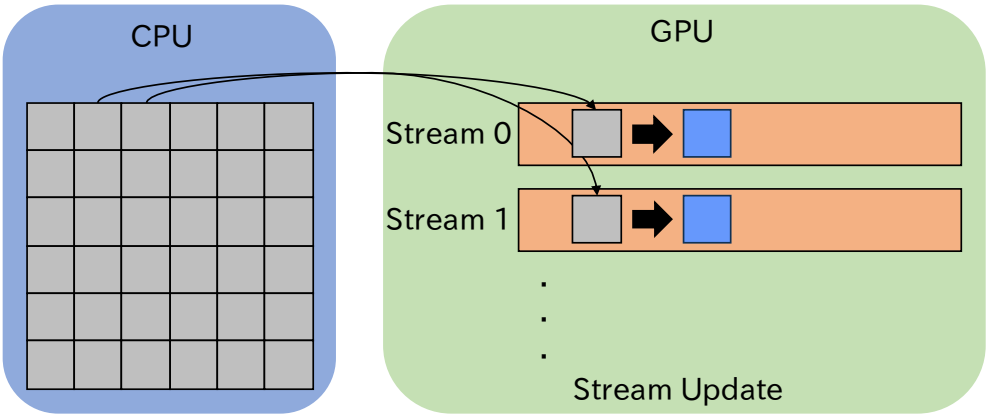

図 7.3: stream update の概要図

反する．これまで，GPU では小さいサイズのカーネル実行が効率的に実行できなかったが，複数カーネルの同時実行ができれば計算資源の稼働率を向上させることが可能 [38, 39, 40] と考えられる．一方で，タイルサイズ分の小さなデータ転送が頻発する．ホストからデバイス，デバイスからホストへのデータ転送は逐次実行であるため，性能を阻害する可能性がある．Bulk Update と Stream Update に関する実験は 8.2 に示す．

7.5 再帰的 QR 分解

GPU で更新カーネルを実行する場合，タイルサイズが大きいほど GPU の計算資源を有効に活用できることと，一方で CPU 側の分解カーネルの実行時間も増大するため負荷分散がうまく行えないことは既に述べた．また，CPU 側でタイルアルゴリズムのカーネルはシングルスレッド（＝1 コア）で実行されるため，マルチコア CPU の計算資源を有効に活用できない．GPU にあわせて大きくなったタイルサイズに対して CPU 側でも効率的に計算する方法として，再帰的 QR 分解という手法を行った．この手法は，図 7.4 の左上タイルのように，CPU が処理を行うタイルを再度タイルに分割を行い，QR 分解を行う事で CPU のタスク量を増やす手法である [41, 42, 43]．再分割を行う事で，look-ahead の実装と同等の効果が得られる．しかし，GPU 側の更新処理も再分割を行ったタイルサイズで更新する必要があるため，GPU の作業量が増えてしまう．再帰的 QR 分解に関する実験は 8.4 に纏めてある．

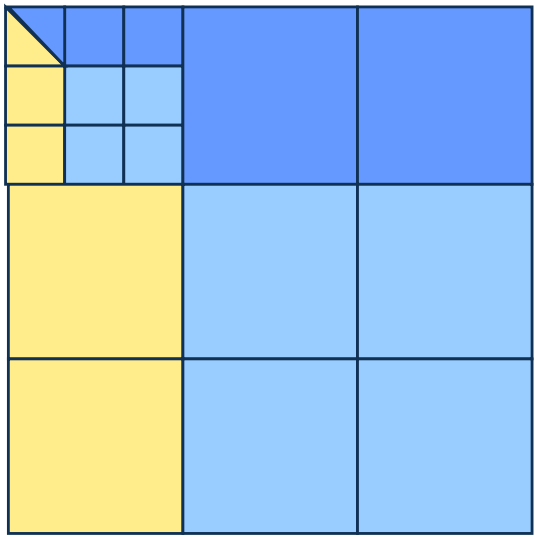

図 7.4: 再帰的 QR 分解

第8章　性能評価

本研究では，マルチコアクラスタシステムにおけるタイルサイズチューニングの計算モデル評価と CPU/GPU クラスタシステムにおけるタイル CAQR アルゴリズムの性能評価を行った．計算モデルの評価には理化学研究所計算科学研究センターが所有する京コンピュータを，CPU/GPU クラスタシステムにおけるタイル CAQR アルゴリズムの性能評価には，東京工業大学学術国際情報センターが所有する TSUBAME 2.5 および東京大学情報基盤センターが所有する Reedbush-H，名古屋大学情報基盤センターが所有する不老 Type II を用いて性能評価を行った．各システムの 1 ノードの構成については，表 8.2，8.1，8.3，8.4 の通りである．

8.1　タイルサイズチューニング

この実験では，京コンピュータを用いてタイルサイズチューニングを行った．1 ノードあたりの行列サイズを 64000 × 8000 の縦長行列で固定し，ノード数が増えるに従って縦方向のサイズを増やす，つまり $64000P \times 8000$ の行列に対してタイルサイズチューニングを行った．図 8.1 はタイルサイズを 40,100,200,400,800 を選択し，ノード数を 2 の累乗に増やしていった時の計算モデルの予想実行時間を示したグラフである．予想実行時間が横ばいなのは，バイナリツリーではノード数が倍に増えても，マージ実行の回数が 1 ステップしか増えないためである．この計算モデルからもっとも速いのはタイルサイズ 400 の時であることが分かる．また，タイルサイズ 400 の時の実実行時間と，性能モデルの予想実行時間の比から，本性能モデルでは約 90%の精度で実際の計算時間が求められることが分かった．

表 8.1: TSUBAME 2.5 specifications

Processor	Intel Xeon X5670 x2
Memory	54 GB
GPU	NVIDIA Tesla K20X x3
GPU Memory (single)	6GB
Compiler	Intel C++ compiler 15.0.2
MPI Library	OpenMPI 1.8.2
CUDA & cuBLAS	7.5

表 8.2: K computer specifications

Processor	SPARC64 VIIIfx 8C 2GHz
Memory	16 GB per node
Open MP	OpenMP3.1
MPI	MPI-2.2

表 8.3: Reedbush-H specifications

Processor	Intel Xeon E5-2695v4 x2
CPU Memory	256 GB
GPU	NVIDIA Tesla P100 x2
GPU Memory (single)	16GB
OS	Red Hat Enterprise Linux 7
Compiler	Intel C++ compiler 17.0.1.132
MPI Library	Intel MPI 2017.1.132
BLAS	Intel MKL 17.1.132
CUDA & cuBLAS	8.0.44
MAGMA Library	2.2.0

表 8.4: 不老 Type II specifications

Processor	Intel Xeon Gold 6230 x2
CPU Memory	384 GB
GPU	NVIDIA Tesla V100 x4
GPU Memory (single)	32GB
OS	CentOS 7.7
Compiler	Intel C++ compiler 2020.1.217
MPI Library	Intel MPI 2020.1.217
BLAS	Intel MKL 2020.1.217
CUDA & cuBLAS	10.2.89

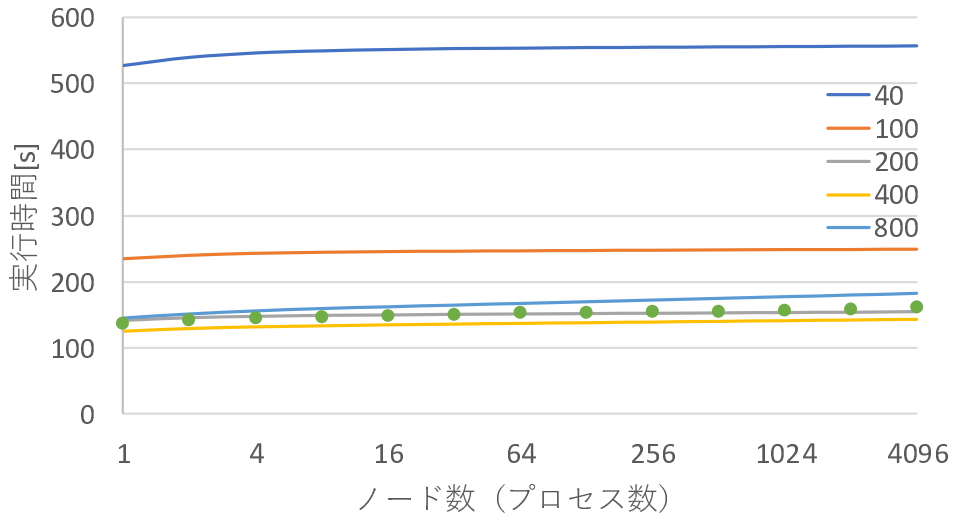

図 8.1: 京コンピュータにおけるバイナリツリー選択時の計算モデルによる実行時間予測と，タイルサイズ 400 の実行時間．実線が計算モデルによる予想実行時間．マーカーが実実行時間．横軸にノード数．縦軸は実行時間．

8.2 1 ノードにおける性能評価

8.2.1 Reedbush-H

以下の実験では，GPU で実行される更新タスクには前述の 2 手法，Bulk Update と Stream Update の手法を用いた．各手法のタイルサイズ，内部ブロック幅ともにタイルサイズチューニングを行い，最適値を使用している．CPU/GPU ヘテロジニアス環境における QR 分解ルーチンとして，MAGMA ライブラリから magma_dgeqrf2_mgpu ルーチンが提供されている．本実験では東京大学情報基盤センターの Reedbush-H 1 ノード使用した，各実装のタイル CAQR アルゴリズムと magma_dgeqrf2_mgpu の実行速度を比較した（図 8.2）．今回行った Bulk Update と Stream Update では Stream Update 実装の方が CPU/GPU ヘテロジニアス環境では実行速度が速かった．これは，OpenMP と CUDA Stream による非同期実行が影響したと考えられる．Stream Update 実装の速度は magma_dgeqrf2_mgpu ルーチンのピーク性能の半分であるが，magma_dgeqrf2_mgpu の約 2 倍のサイズの行列を分解することができる．MAGMA は全行列データを GPU メモリ上に保存し，CPU で実行される分解タスクに必要な行列データをその都度メインメモリ上に転送する方式を取る．しかし，我々の実装では個々の更新タスクに対して毎回 CPU-GPU 間の転送が必要であるため，MAGMA よりも多くのデータ転送が必要となる．そのため，MAGMA が処理できる行列サイズでは非常に大きな性能差が生じてしまう．

また，性能差に関する調査として，行列サイズ 30720 の正方行列に対する Bulk Update と Stream Update の GPU 側の実行時トレースを行った．トレース結果は，図 8.3 となる．図の Memcpy のラインがデータ通信部分であり，Compute 部分が GPU での計算実行状況を示している．Bulk Update と Stream Update を比較すると，Stream Update は複数のカーネルが同時実行されており，計算が多く実行されているのが分かる．

また，図 8.4 は Bulk Update における double buffering の有無による性能差についてのグラフである．double buffering を行う事で GPU メモリを 2 倍消費するが，通信の隠蔽が行われ，性能が向上することが分かる．

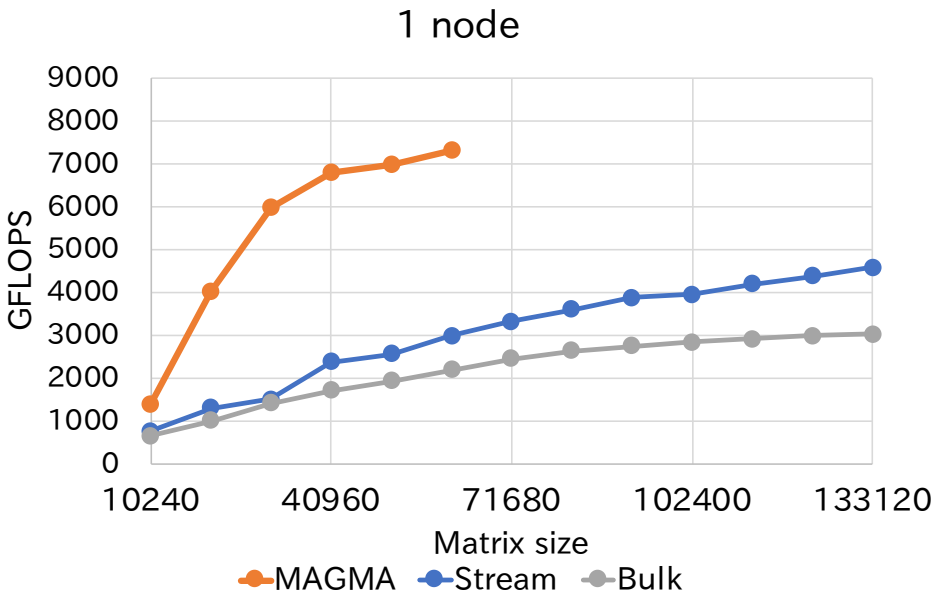

図 8.2: Bulk Update, Stream Update 手法および MAGMA の性能測定結果．Reedbush-H において正方行列に対して QR 分解を 5 回計測を行い，最大・最小を除いた 3 点の平均実行時間．行列の値はメルセンヌ・ツイスター法による乱数から生成．横軸は正方行列の行列サイズ，縦軸は一秒あたりの浮動小数点演算回数．

8.2.2 不老 type II サブシステム

次に，名古屋大学情報基盤センターの不老 Type II サブシステムを用いて GPU の台数を変化させた時の性能評価を行った．Bulk Update および，Stream Update では最適なタイルサイズおよび，内部ブロック幅のパラメータチューニングを行っている．Bulk Update，Stream Update による性能結果は図 8.5 となる．1GPU の時に非常に大きな性能差が生じている．これは，1GPU 時はタイル QR 分解であり，Stream Update では j 方向が並列に実行されるため，深い Look-ahead が行われるためである．また，4GPU 使用した時では Stream Update が Bulk Update よりも 1TFLOPS ほど高いのが読み取れる．これは Reedbush-H と同様に Open MP と CUDA Stream による非同期実行が影響していると考えられる．

性能差に関する調査として，行列サイズ 40960 の正方行列に対する Bulk Update と Stream Update の 4GPU 使用時の GPU 側の実行時トレースを行った．トレース結果は，図 8.6 となる．前述のトレース図と同様に，Memcpy のラインがデータ通信部分であり，Compute 部分が GPU での計算実行状況を示している．Bulk Update と Stream Update を比較すると，Stream Update は複数のカーネルが同時実行されており，計算が多く実行されているのが分かる．

また，周期的に現れる空白部分は CAQR アルゴリズムのマージ処理のための通信待ちと考えられる．プロセス数を増やすほど，空白部分は増える．このアイドル状態を解消させれば，より高速化が行われると考えられる．

図 8.2 と図 8.5 の 2 ノードを比較すると，ピーク性能にそれほど差が無いことが読み取れる．このことから，GPU の性能を十分に引き出せていないか，CPU で行われる分解ルーチンがボトルネックとなっていると考えられる．

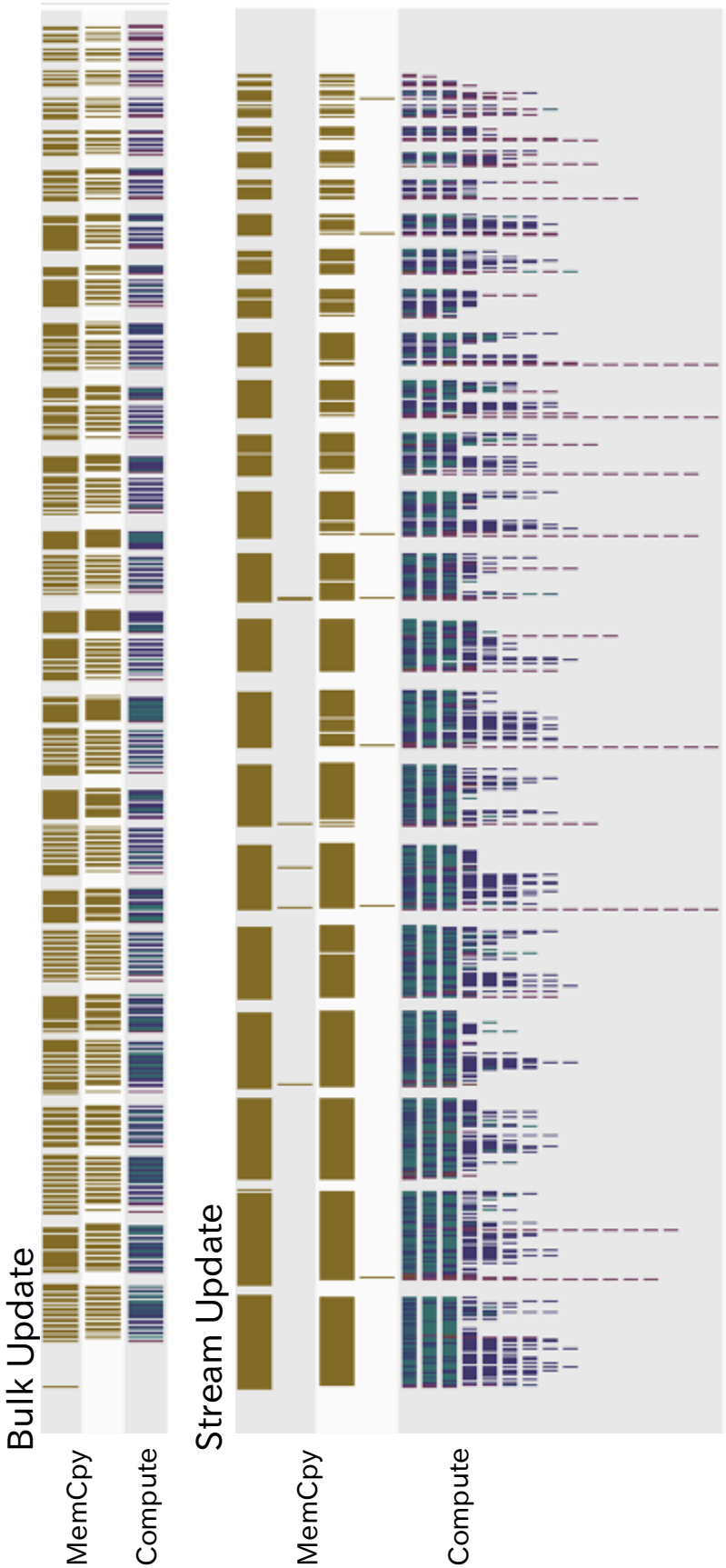

図 8.3: Bulk Update, Stream Update 手法について Reedbush-H 1 ノードにおける GPU トレースの結果.

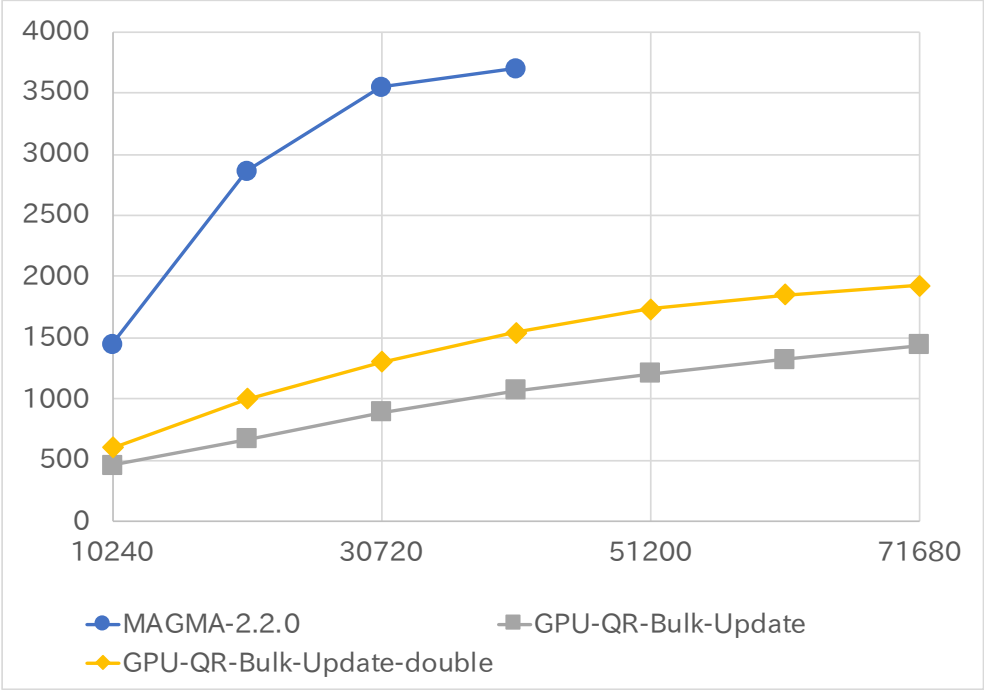

図 8.4: Bulk Update の double buffering の有無による性能差．Reedbush-H において実行．行列の値はメルセンヌ・ツイスター法による乱数から生成．横軸は正方行列の行列サイズ，縦軸は一秒あたりの浮動小数点演算回数．

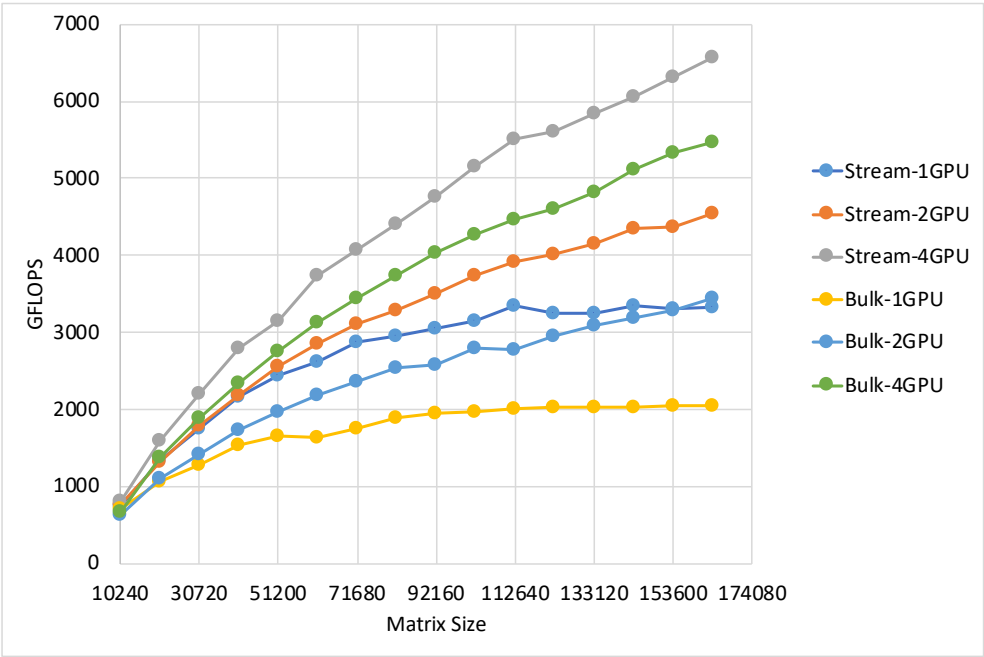

図 8.5: 不老 Type II における Bulk Update, Stream Update 手法の台数による性能測定結果．正方行列に対して QR 分解を行った実行時間．行列の値はメルセンヌ・ツイスター法による乱数から生成．横軸は正方行列の行列サイズ，縦軸は一秒あたり浮動小数点演算回数．

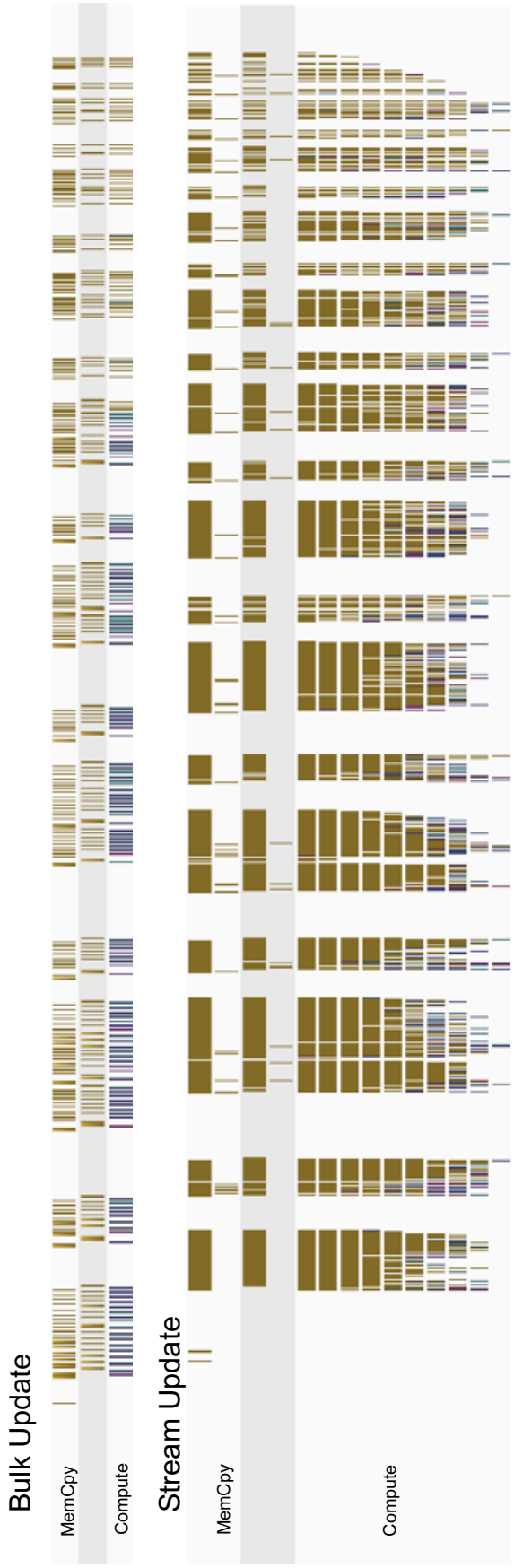

図 8.6: Bulk Update, Stream Update 手法について不老 Type2 1 ノードにおける GPU トレースの結果.

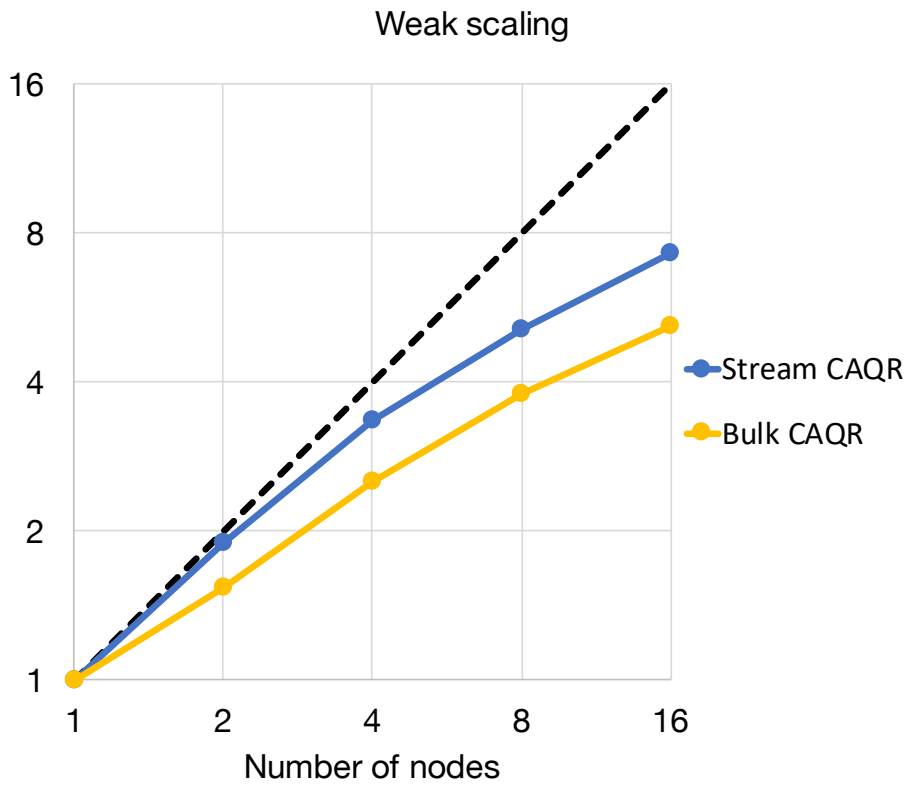

図 8.7: Reedbush-H における Bulk Update, Stream Update 手法の Weak Scaling の結果．1 ノードあたりの行列サイズを 81290 × 81920 に固定して台数効果を測定．横軸は使用ノード数，縦軸は 1 ノードの性能を 1 とした時の並列化効率．

8.3 並列化効率

Reedbush-H において Stream Update と Bulk Update の Weak Scaling の性能比較を行った．Weak Scaling とは，1 ノードあたりの問題サイズを固定して性能評価を行い，ノード数を増加させた時の台数効果を調べる指標である．今回は 1 ノードあたりの行列サイズを 81920 × 81920 とし，ノード数を 1 ノードから 16 ノードまで増加させた時の台数効果を測定した．行列データは 1D ブロックサイクリックデータ分散を用いて分散させており，各プロセスには行列データをタイル行毎にサイクリックに分散している．測定結果を図 8.7 に示す．2 種類の実装のうち Stream Update 実装の方が高い並列化効率を示した．これは，1 ノードの実行結果と同様に非同期実行が影響していると考えられる．

次に，Reedbush-H において Stream Update と Bulk Update の Strong Scaling の性能比較を行った．Strong Scaling とは，並列計算で扱う問題サイズを固定し，並列数を増やしていくことで台数効果を調べる指標である．並列数を増やすにつれて，1 計算装置が処理する問題サイズが減少するが通信回数は変わらないため，一般に並列効果は出にくいとされている．本実験では，行列サイズを 102400 × 102400 に固定し，ノード数を 1 ノードから 16 ノードまで増加させた時の Strong Scaling の台数効果を測定した．測定結果を図 8.8 に示す．行列データは weak scaling と同様に 1D ブロックサイクリックデータ分散を用いた．この実験においても Stream Update 実装の方が高い効率となったが，十分な並列化性能とは言えない．ノード数が増加した時，1 プロセスあたりのドメイン数が小さくなりドメイン内タイル QR 分解は並列化されるが，ドメイン間のマージ処理回数が増え，最上位ドメインのマージがボトルネックとなることが並列化効率の阻害要因である．

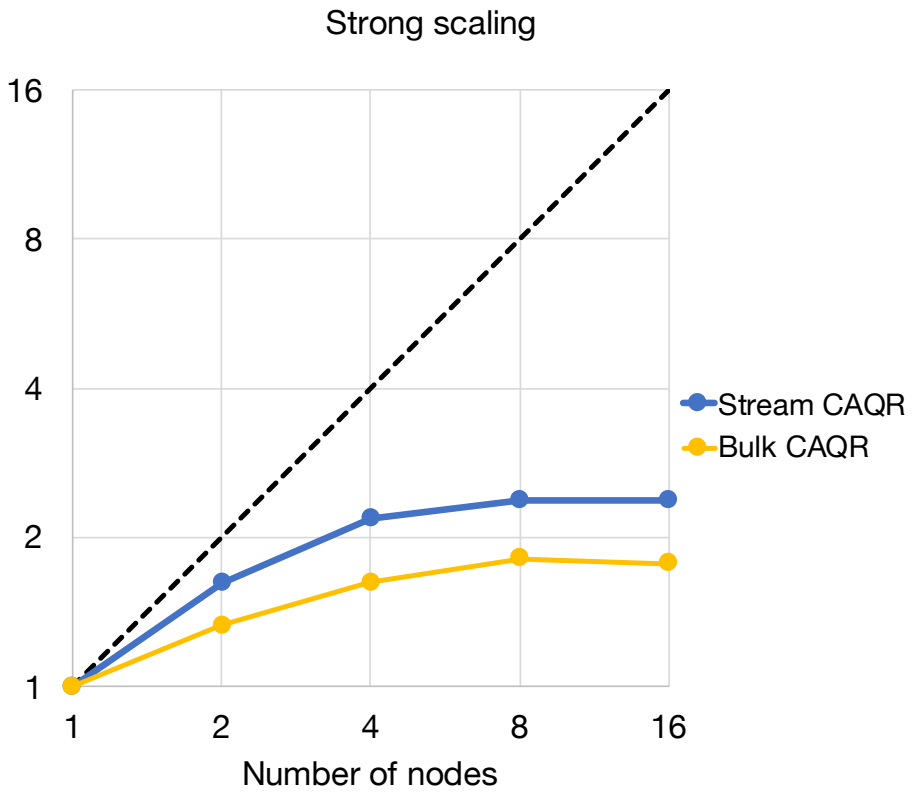

図 8.8: Reedbush-H における Bulk Update, Stream Update 手法の Strong Scaling の結果. 行列サイズを 102400 × 102400 に固定して台数効果を測定．横軸は使用ノード数，縦軸は 1 ノードの性能を 1 とした時の並列化効率.

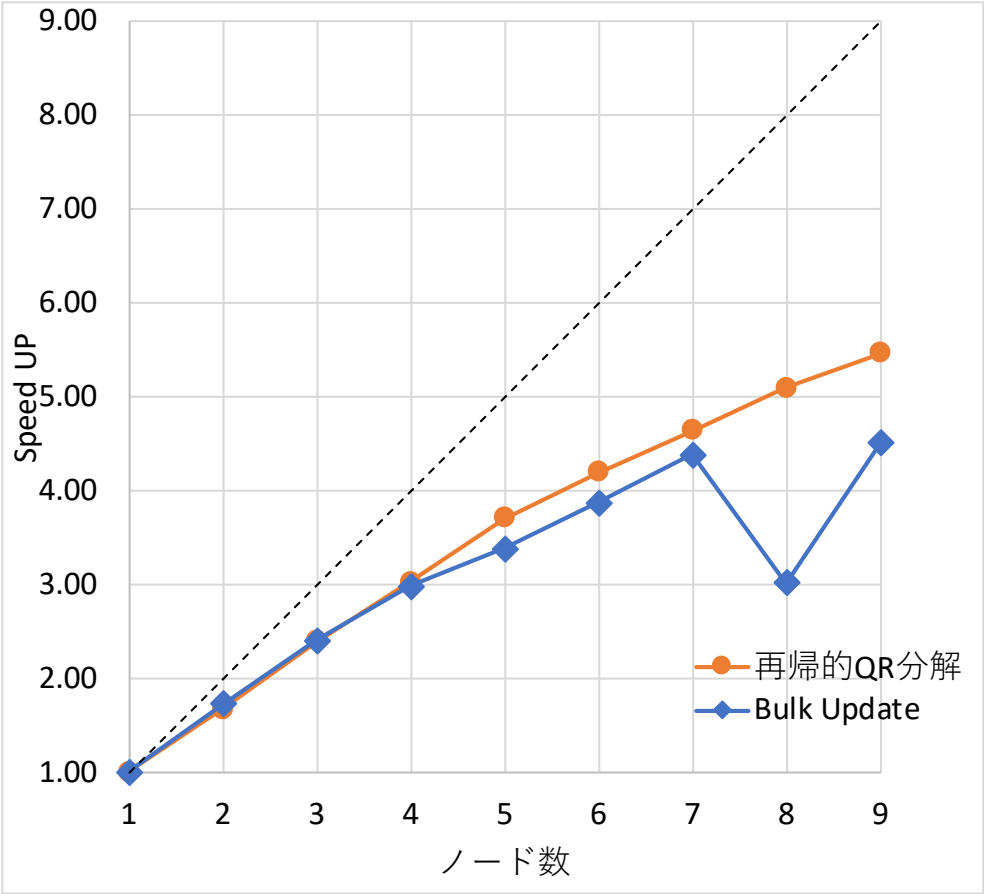

図 8.9: TSUBAME 2.5 における再帰的タイル QR 分解と Bulk Update の性能測定．1 ノードあたりの行列サイズを 40960 × 40960 に固定して台数効果を測定．横軸は使用ノード数，縦軸は 1 ノードの性能を 1 とした時の並列化効率．

8.4 再帰的タイル QR 分解の性能測定

図 8.9 は TSUBAME 2.5 における再帰的タイル QR 分解と Bulk Update の性能測定の結果である．5 ノードあたりまでは比較的良くスケーリングしているが，さらにノード数が増加すると効率が落ち，9 ノード時で約 60%となる．再帰的タイル QR 分解では，ノード数が増加すると各プロセスの GPU が更新処理を行うタイル列が少なくなるため，GPU の性能を十分に発揮させる事ができない．また，分解カーネルの i 方向処理が逐次的であるため，ボトルネックとなってしまっていると考えられる．

第9章　結論

本論文は，近年の高並列計算機として定着しつつあるCPU/GPUヘテロジニアスクラスタシステムに適する数値線形代数ライブラリの開発ため，QR分解のタイルアルゴリズムによる高性能化および，性能モデルによるタイルサイズチューニングを試みた．QR分解を含む数値線形代数ライブラリは，さまざまな並列計算機環境のために開発が行われているが，CPU/GPUヘテロジニアスクラスタシステム向けのライブラリ開発は行われていない．そこで，本研究では大規模並列環境における通信量削減を行ったCAQRアルゴリズムを用いる事で，ノード間の通信量削減を行うと共に，並列性の向上を行った．また，GPUにおける処理の実装方法としてBulk UpdateとStream Updateの2手法を考案，実装し性能評価を行った．その結果，Reedbush-Hで行った実験では，Stream UpdateではWeak Scalingで16ノード使用時で1ノード時の約8倍の性能が得られたが，Strong Scalingでは16ノード使用時で1ノード時の約3倍となった．Stream Update手法は複数の更新カーネルを並列に実行可能であり，計算と通信のオーバーラップが行われるため，今回行った2種類の並列化効率についても，高い性能が得られたと考えられる．また，タイルアルゴリズムの「細粒度のタスクを大量に生成し，非同期に実行を行う」という特徴をGPUでも生かせることを示した．

タイルアルゴリズムにおいては，タイルサイズのチューニングが必須であり，広範囲なパラメータ空間の探索を行う必要があった．そこで，本研究では3種類のカーネルについてのみ計測し，マルチコアクラスタシステムにおける性能モデルを作成する事で，実行時間の予測を行えるモデルを作成した．作成した性能モデルにより，3種類のカーネルの実行時間を求めるだけで，誤差約10%の範囲で実行時間を求める事が可能となった．

本研究の手法では，主要な処理をGPUで実行することで性能向上が行われている一方で，CPUの計算資源の活用が十分とは言えない．これについては，CPU側に後続行列更新処理の一部を割り当てるなどの方法が考えられる．今後，さらなる性能向上を行うには，CAQRにおけるマージ処理をバイナリツリー形式以外，例えばフラットツリーやGreedyアルゴリズムなどの手法を用いることが考えられる．また，現在の実装ではNVIDIAのCUDAライブラリに依存している．近年，AMD社もGPU製品の開発に力を入れ始めており，またIntel社もdiscrete GPUの開発を進めている．NVIDIA社のGPUは近年，機械学習向けの機能を強化しており，それらを科学技術計算分野でどのように活用するかを検討することも課題としてあげられる．今後さまざまなヘテロジニアス環境が提供される可能性も考えると，OpenCLやOpenACCなどによる非環境依存なプログラム開発への適応も考える必要がある．

謝辞

本研究を進めるにあたり，様々な御指導をしてくださいました鈴木智博准教授に心から感謝を申しあげます.

参考文献

[1] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "Parallel tiled QR factorization for multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 13, pp. 1573 – 1590, (2008).

[2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorense, *LAPACK's user's guide, 3rd. edition*. Philadelphia: SIAM, 1999.

[3] PLASMA (2021). [Online]. Available: https://bitbucket.org/icl/plasma.

[4] OpenBLAS (2021). [Online]. Available: https://https://www.openblas.net.

[5] TOP500 (2021). [Online]. Available: https://www.top500.org.

[6] R. James, SC13 Tutorial silde. (2013). [Online]. Available: https://software.intel.com/content/www/us/en/develop/blogs/structure-parallel-programming-tutorial-materials-posted.html.

[7] S. Fuller and L. Millett, *The future of computing performance: Game over or next level?*, National Research Council, (2011), Washington DC, The National Academies Press.

[8] H. Sutter, *The Free Lunch is Over*, Dr. Dobb's Journal, 30 (3), 2005.

[9] Netlib Repository. (2021). [Online]. Available: http://www.netlib.org.

[10] S. Blackford and others, *ScaLAPACK user's guide*, SIAM, 1997.

[11] G. Bosilca and A. Bouteiller and A. Danalis and M. Faverge and H. Haidar and T. Herault and J. Kurzak and J. Langou and P. Lemarinier and H. Ltaief and P. Luszczek and A. YarKhan and J. Dongarra, Distributed-Memory Task Execution and Dependence Tracking within DAGuE and the DPLASMA Project, LAPACK Working Note 232, UT-CS-10-660, 2010.

[12] MAGMA (2021). [Online]. Available: http://icl.cs.utk.edu/magma/.

[13] M. Flynn, Some Computer Organizations and Their Effectiveness, IEEE Trans. Comput., Vol. C-21, pp. 948 (1972).

[14] 森正武, 数値解析 第 2 版, (2002), 東京, 共立出版.

[15] H. Bischof, Adaptive blocking in the QR factorization, The Journal of Supercomputing, No. 3, Vol. 3, pp. 193 – 208, 1989.

[16] J. Dongarra and S. Ostrouchov, LAPACK Block Factorization Algorithms on the Intel iPSC/860, LAPACK Working Note 24, UT-CS-90-115, 1990.

[17] R. Schreiber and C. Van Loan, A storage-efficient WY representation for products of Householder transformations, SIAM J. Sci. Statist. Comput., Vol. 10, Num. 1, pp. 52 – 57, (1989).

[18] A. Buttari and J. Langou and J. Kurzak and J. Dongarra, Parallel tiled QR factorization for multicore architectures, Concurrency and Computation: Practice and Experience, No. 13, Vol. 20, pp. 1573 – 1590, 2008.

[19] A. Buttari and J. Langou and J. Kurzak and J. Dongarra, A class of parallel tiled linear algebra algorithms for multicore architectures, Parallel Comput., Vol. 35, No. 11, pp. 38 – 53, 2009.

[20] J. Kurzak and J. Dongarra, QR Factorization for the CELL Processor, Scientific Programming, Special Issue: High Performance Computing with the Cell Broadband Engine, Vol. 17, No. 1-2, pp. 31 – 42, 2009.

[21] F. Song and H. Ltaief and B. Hadri and J. Dongarra, Scalable Tile Communication-Avoiding QR Factorization on Multicore Cluster Systems, 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10), pp. 1 – 11, (2010).

[22] A. YarKhan and J. Kurzak and J. Dongarra, QUARK Users' Guide: QUeueing And Runtime for Kernels, University of Tennessee Innovative Computing Laboratory Technical Report, ICL-UT-11-02, (2011).

[23] M. Takayanagi and T. Suzuki, Construction of performance model of tile CAQR and performance result of the implementation, Proceedings of IEEE 11th International Symposium on Embedded Multicore/Many-core SoCs (MCSoC-17), pp.151 157, 2017/09/20.(6 章の内容に関連する)

[24] J. Demmel and L. Grigori and M. Hoemmen and J. Langou, Communication-avoiding parallel and sequential QR and LU factorizations, SIAM J. Sci. Comput., Citeseer, 2008.

[25] J. Demmel and L. Grigori and M. Hoemmen and J. Langou, Communication-optimal parallel and sequential QR and LU factorizations, SIAM J. Sci. Comput., Vol. 34, No. 1, pp. A206 – A239, 2012.

[26] E. Agullo and C. Coti and J. Dongarra and T. Herault and J. Langou, QR factorization of tall and skinny matrices in a grid computing environment, Proceedings of IPDPS ' 10, the 24st IEEE Int. Parallel and Distributed Processing Symposium, 2010.

[27] F. Song and H. Ltaief and B. Hadri and J. Dongarra, Scalable Tile Communication-Avoiding QR Factorization on Multicore Cluster Systems, 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC '10, 2010.

[28] 高柳雅俊, 鈴木智博, 縦長行列におけるタイル CAQR アルゴリズムの性能評価, 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）, 2017-HPC-158 (23), pp. 1–8, 2017/03/01.

[29] 鈴木智博, 高柳雅俊, 縦長行列に対するマルチコアクラスタ向け QR 分解アルゴリズム, 日本計算工学会第 22 回計算工学講演会（ソニックシティ）2017/06/02.

[30] 鈴木智博, 高柳雅俊, クラスタ環境向けタイル TSQR の性能モデル, 日本応用数理学会行列・固有値研究会 in SWoPP 秋田, 2017/07/27.

[31] T. Suzuki and M. Takayanagi, Implementation of tile matrix factorization and its performance model, Conference on advanced topics and auto tuning in high-performance scientific computing (ATAT2018), 2018/03/26.

[32] 鈴木智博, 高柳雅俊, タイルサイズチューニングのためのタイル QR アルゴリズムの性能モデル, 日本応用数理学会 2018 年度年会（名古屋大学）2018/09/05.

[33] M. Anderson and G. Ballard and J. Demmel and K. Keutzer, Communication-avoiding QR decomposition for GPUs, Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International, pp. 48 – 58, IEEE, 2011.

[34] J. Kurzak and R. Nath and P. Du and J. Dongarra, An implementation of the tile QR factorization for a GPU and multiple CPUs, PARA'10: State of the Art in Scientific and Parallel Computing, Reykjav?k, Iceland, 2010, 2010.

[35] 高柳雅俊, 鈴木智博, ヘテロジニアスクラスタシステムにおけるタイル CAQR アルゴリズム実装, 日本応用数理学会行列・固有値研究会 in SWoPP 秋田, 2017/07/27.

[36] 高柳雅俊, 鈴木智博, マルチ GPU 環境におけるタイル CAQR アルゴリズムの実装, 日本応用数理学会 2014 年度年会（政策研究大学院大学）2014/09/05.

[37] T. Suzuki and M. Takayanagi and K. Araki, Implementation of tile algorithms for matrix decomposition on CPU-GPU system, Conference on advanced topics and auto tuning in high-performance scientific computing (ATAT2015), 2015/02/27.

[38] M. Takayanagi and T. Suzuki, Communication-Avoiding tile QR decomposition on CPU/GPU heterogeneous cluster system, Proceedings of IEEE 12th International Symposium on Embedded Multicore/Many-core SoCs (MCSoC-18), pp.125 131, 2017/09/20.(7 章の内容に関連する)

[39] M. Takayanagi and T. Suzuki, Dynamic task scheduling implementation of tile QR decomposition on CPU/GPU heterogeneous cluster system, SIAM Conference on Parallel Processing for Scientific Computing, (Poster presentation), 2018/03/08.

[40] 高柳雅俊, 鈴木智博, CPU/GPU クラスタシステムにおけるタイル QR 分解の実装と性能評価, 日本計算工学会第 23 回計算工学講演会（ウインクあいち）2018/06/07.

[41] 高柳雅俊, 鈴木智博, CPU/GPU 混在環境における再帰的タイル QR 分解, 日本応用数理学会 2015 年度年会（金沢大学）2015/09/11.

[42] 高柳雅俊, 鈴木智博, CPU/GPU 混在環境における再帰的タイル QR 分解の動的スケジューリング実装, 日本応用数理学会 2016 年度研究部会連合発表会（神戸学院大学）2016/03/04.

[43] 高柳雅俊, 鈴木智博, クラスタ型ヘテロジニアス環境におけるタイル QR 分解, 日本応用数理学会 2016 年度年会（北九州国際会議場）2016/09/13.