6J-04

CPU-GPU 環境におけるブロック QR 分解のタスク並列化 *

霜鳥竜輝 † 鈴木智博 ‡
山梨大学大学院 §

1 はじめに

QR 分解とは，$m \times n$ $(m \geq n)$ の行列 A を $m \times m$ の直交行列 Q と $m \times n$ の上三角行列 R に分解することであり，最小二乗法や特異値分解の前処理などで使われる．QR 分解は計算量が多く，計算速度の向上が求められる．そのため，本研究では QR 分解を CPU-GPU システム上で実行することによって，高速な QR 分解を実現することを目的としている．

2 ブロック QR 分解

2.1 fork-join 型アルゴリズム

QR 分解のアルゴリズムの 1 つにブロック QR 分解がある．このアルゴリズムは行列 A を列方向に分割して複数の部分行列 (ブロック) $A_1, A_2, \cdots$ を生成し，ブロックごとに式 (1) のパネル分解と式 (2) の後続ブロックの更新を実行する．

$$R_i, V_i, T_i \leftarrow A_i \tag{1}$$

$$A_j \leftarrow (I - V_i T_i V_i^T) A_j \quad (j > i) \tag{2}$$

パネル分解，後続ブロックの更新を実行するルーチンはそれぞれ分解ルーチン，更新ルーチンと呼ばれる．ブロック QR 分解では，分解ルーチンは逐次的な処理が多く，L1，L2BLAS ルーチンで構成され，更新ルーチンは L3BLAS ルーチンで構成されている．そのため，分解ルーチンを CPU，更新ルーチンを GPU で実行することで CPU，GPU 双方の利点を活用できる．

しかし，ブロック QR 分解は更新ルーチンが fork 部分に，分解ルーチンが join 部分に相当する fork-join 型の並列実行モデルであるため，マルチプロセッサの利点を発揮させることができないアルゴリズムである．

2.2 タスク並列型アルゴリズム

タスク並列型実行モデルは大量の細粒度タスクを非同期に実行することで計算資源を有効活用することを目的としており，並列度の高いアーキテクチャ向きであるとして近年注目されている．

ブロック QR 分解のタスク並列化は，更新ルーチンを一度に実行するのではなく，細かく分けて分解ルーチンと並列に実行する．これを行うことで，図 1 中の $R_{i+k}, V_{i+k}, T_{i+k} \leftarrow A_{i+k}$ と $A_{i+k+1} \leftarrow (I - V_i T_i V_i^T) A_{i+k+1} \cdots$ を並列に実行することができる．

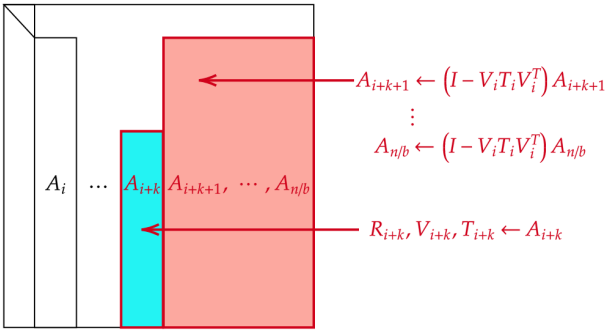

図 1 タスク並列型ブロック QR 分解における並列化可能な部分

このように，タスク並列化を行うことで V_i, T_i による更新が完了する前に $R_{i+k}, V_{i+k}, T_{i+k} \leftarrow A_{i+k}$ を実行できる．これを look-ahead と呼ぶ．look-ahead によって，分解ルーチンを先に実行することで実行可能な更新ルーチンの数を増やすことができる．すなわち，look-ahead の利点は並列実行可能なタスクを増やすことである．

3 タスク並列型ブロック QR 分解の実装

OpenMP には，タスク並列を実現するタスク構文 (task) が存在する．タスク構文は，逐次プログラムにディレクティブを挿入するだけでタスク並列化が実装可能である．実行するタスクに依存関係が存在する場合，depend 節を用いてそれを記述する．また，タスク並列化を行うと実行可能なタスクが複数生成されることがあるが，それらのタスクを実行する優先度をタスク構文の priority 節で決めることができる．タスク構文を用いたブロック QR 分解のタスク並列化に関する 4 つの手法を次に示す．なお，このセクションで説明する実行結果はすべて CPU 上のみで実行したものである．

---

* Task parallelization of block QR decomposition on CPU-GPU computing environment
† Ryuki Shimotori
‡ Tomohiro Suzuki
§ University of Yamanashi

3.1 手法 (1)

手法 (1) は，タスクに優先度をつけずにタスク並列化を行う方法である．優先度をつけない場合，先に実行可能になったタスクをタスクキューに入れることになる．例えば，$R_i, V_i, T_i \leftarrow A_i$ が完了した場合，先に実行可能になるのは $R_{i+1}, V_{i+1}, T_{i+1} \leftarrow A_{i+1}$ ではなく $A_{i+1} \leftarrow (I - V_i T_i V_i^T) A_{i+1}, A_{i+2} \leftarrow (I - V_i T_i V_i^T) A_{i+2} \cdots$ であるため，後者が優先的に実行される．この場合だと，更新ルーチンが完了しないと次の分解ルーチンを実行できないため，この手法では look-ahead ができない．これを解決するためには，各タスクに優先度をつける必要がある．

3.2 手法 (2)

手法 (2) では，分解ルーチンを優先的に実行するために，分解ルーチンの優先順位を更新ルーチンよりも高くした．この場合，例えば $R_i, V_i, T_i \leftarrow A_i$ が完了した場合，そのすぐ後に $R_{i+1}, V_{i+1}, T_{i+1} \leftarrow A_{i+1}$ が実行された．しかし，$R_{i+2}, V_{i+2}, T_{i+2} \leftarrow A_{i+2}$ の実行は $A_{i+1} \leftarrow (I - V_i T_i V_i^T) A_{i+1}, A_{i+2} \leftarrow (I - V_i T_i V_i^T) A_{i+2} \cdots$ が完了するまで行われない．これでは look-ahead の深さが少ないため，look-ahead の利点が発揮されない．

3.3 手法 (3)

手法 (3) では，タスクの優先順位を図 2 のように割り当てた．

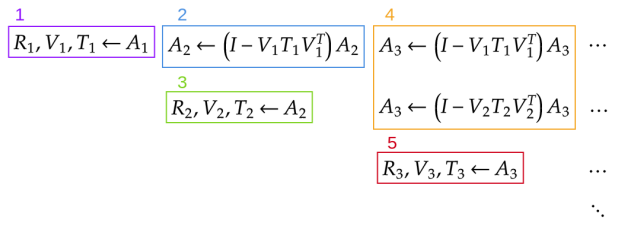

図 2 手法 (3) のタスクの優先順位

手法 (3) によって，分解ルーチンが連続で実行され，深い look-ahead が実現できた．しかし，スレッドごとに終了時刻が異なり，処理の終盤では一部のスレッドが遊休状態になった．この原因は，優先度の過剰な割り振りによって処理の終盤では並列に実行できないタスクが残るためである．

3.4 手法 (4)

手法 (3) を使っても，look-ahead はある程度の深さまでしか行われないことがわかった．そのため，手法 (4) ではその上限に達したら優先度の割り振り方を切り替える方法をとった．このために，まずはこの上限における分解ルーチンの実行回数 u を求める必要がある．

今回は，重回帰分析を用いて u を推定することにした．その時の説明変数には行列サイズ・ブロックサイズを用いたが，行列サイズは説明変数として適切でないという帰無仮説を棄却できなかったため，説明変数から除外した．

手法 (4) では $A_1 \sim A_u$ までのルーチンを手法 (3) の優先度の割り振り方，A_{u+1} 以降のルーチンを手法 (2) の優先度の割り振り方という形で，途中で優先度の割り振り方を切り替える．

手法 (4) を実行した結果，分解ルーチンが連続で実行されていて，かつすべてのスレッドが同じ時刻に終了した．

4 実行時間

手法 (1)〜手法 (4) の計算時間を表 1 に示す CPU を用いて測定した．その結果を表 2 に示す．なお，表 2 の計算時間もすべて CPU 上のみで実行したものである．また，実験に使用した行列のサイズは 5000×5000 と 10000×10000，ブロックサイズは 128 である．

表 1 実験に使用した CPU のスペック

モデル名	Intel Core i7-920
コア数	4
クロック数	2.67[GHz]
メモリ	24[GB]

表 2 手法 (1)〜手法 (4) の計算時間 [秒](n:正方行列の次元数)

アルゴリズム	$n = 5000$	$n = 10000$
手法 (1)	5.18	38.77
手法 (2)	4.86	37.28
手法 (3)	5.03	38.25
手法 (4)	4.80	37.13

表 2 より，計算時間は手法 (4) < 手法 (2) < 手法 (3) < 手法 (1) となった．しかし，手法 (4) と手法 (2) では実行時間にほとんど差がないことがわかる．これは，手法 (4) は処理の序盤以外は手法 (2) と同じ内容であるからだと考えている．

5 今後の課題

まずは，手法 (4) を CPU-GPU 上で実行して性能を評価・考察する必要がある．また，手法 (4) は look-ahead の恩恵が処理の序盤でしか受けられないため，処理の中盤以降でも look-ahead の恩恵が受けられるようなアルゴリズムの開発が必要である．それと，行列サイズごとの最適なブロックサイズのチューニングを行わなければ最適解が出せないため，それも行う必要がある．