Vol.2019-HPC-170 No.26
2019/7/25

Batched Kernel APIを用いるOpenMPタスク生成

李 珍泌[1,a)] 渡部 裕[†1] 佐藤 三久[1,†1]

概要：従来の数値計算を行う HPC アプリケーションや近年注目を集めているディープラーニングの分野では独立した小さい計算カーネルを高いスループットで並列処理する技術が活発に研究されている。BLAS カーネルの並列処理を簡単に記述するため、複数の計算カーネルをまとめて一つの API で処理する Batched Kernel API が提案されている。GPU などのメニーコアアーキテクチャは高い電力効率を達成できるが、Batched Kernel API を使うには手動のコード変換が必要である。本研究では OpenMP のタスクプログラミングモデルで Batched Kernel API を利用するコード変換技術の提案を行う。ユーザは数値計算のカーネルを *task* 指示文で記述する。処理系にはバッチ化できる API が事前に登録されており、そのような計算カーネルが複数並列に実行できるときはバッチ化を行う。NVIDIA GPU 向け cuBLAS を用いてバッチ化を行うプロトタイプ処理系を実装した。DGEMM カーネルを呼び出すタスクを処理系が検知し、Batched DGEMM API に変換する。Blocked Cholesky Decomposition を用いた性能評価では処理系によって複数の DGEMM カーネルがバッチ化されることが確認できた。DGEMM のみの計算性能は行列 4096×4096、小行列 128 のケースで逐次実行の 4 倍に高速化されるが、OpenMP のタスクやバッチの処理のオーバーヘッドのため、全体性能は 36%向上する。

OpenMP Task Generation for Batched Kernel APIs

Jinpil Lee[1,a)] Yutaka Watanabe[†1] Mitsuhisa Sato[1,†1]

1. はじめに

数値計算の問題のいくつかは計算カーネルをより小さいものに分割することができる。例えば、行列積は小行列同士の行列積を足し合わせることでより小さい複数の行列積に分割される。近年ディープラーニングを含む機械学習が注目されるようになり、人工知能のアプリケーションでは小さいサイズの計算カーネルをたくさん計算することが求められるものもある。High Performance Computing (HPC) の分野では独立した計算カーネルを高いスループットで処理する技術が重要になってきている。その結果、メニーコアアクセラレータなどの様々なハードウェアをターゲットにして効率の良い計算を行うための研究が活発になってきている。

GPU などのアクセラレータで代表されるメニーコアアーキテクチャは HPC 分野で広く使われるようになり、計算コア数を増やすことでエネルギー効率のいい計算機を実現することができる。Basic Linear Algebra Subprograms (BLAS) に代表される高性能数値計算ライブラリは対象ハードウエアに向けて高度に最適化されており、それらを用いることでハードウエアアーキテクチャの違いを気にすることなく効率のいい計算をプログラムする。しかし、各計算カーネルはメニーコアアーキテクチャを活用するために高い並列性を要求する傾向があり、小さいデータサイズではその性能を十分に発揮することができない。

Batched Kernel Application Programming Interface (API) は小さい計算カーネルをまとめてバッチと呼ばれる一つの大きい計算カーネルとして処理するプログラミング手法である。問題の並列性を高めることで対象ハードウエアの性能を活用することができるため、処理のスループットを向上させることができる。メニーコアアーキテク

1 理化学研究所 計算科学研究センター
RIKEN R-CCS Center for Computational Science
†1 現在，筑波大学 大学院 システム情報工学研究科
Presently with Uniersity of Tsukuba Graduate School of Systems and Information Engineering
a) jinpil.lee@riken.jp

ⓒ 2019 Information Processing Society of Japan

チャ向けの Batched Kernel API は複数の計算コアにバッチ内の計算カーネルをスケジュールすることで計算資源を活用し、高いスループットを実現する。Batched Kernel API を使うためには従来の数値計算 API からのコード変換が必要である。例えば、BLAS の複数の DGEMM カーネルをバッチ化したいときは対象データのポインターを一つの配列にまとめて Batched DGEMM 関数を呼び出す。数値計算ライブラリの Batched Kernel API には標準的な仕様がまだ定まっておらず、対象アーキテクチャ毎に異なる API が提供されている。したがって、コード変換の方法は対象ハードウエアやベンダー毎に異なる。

本研究の目的は従来の数値計算ライブラリの API からそれに対応する Batched Kernel API の関数呼び出しを生成するコード変換技術を提案することである。プログラミングモデルとして OpenMP のタスクプログラミングモデルを採用し、計算カーネルの並列性を記述する。OpenMP はスレッドレベル並列化を記述するプログラミングモデルとして広く使われている。初期の言語仕様は *parallel for* 指示文などを用いてデータ並列化を記述し、対象ハードウエアの並列性を活用するものであった。メニーコアアーキテクチャの登場によって計算コアの数が増え、スレッド間の同期コストが増大するだけでなく、不規則なワークロードの並列化が難しくなってきた。このような問題を解決するために、OpenMP 3.0 では動的なタスクの生成が導入され、不規則なワークロードの並列化が可能になった。OpenMP 4.0 からは *depend* 節によってタスク間の依存性が記述できるようになり、細粒度なスレッド間同期によって同期コストの問題を解決できると期待されている。

本稿は以下のように構成されている。第 2 章は関連研究について述べる。第 3 章は Batched BLAS の概要を説明し、GPU 上での実行モデルや OpenMP との併用について述べる。第 4 章では本研究の目的である OpenMP タスクプログラミングモデルを用いた Batched Kernel API の生成手法について述べる。処理系によるコード変換やランライムライブラリによるタスクスケジュールについて説明を行う。第 5 章では NVIDIA GPU と cuBLAS を用いた性能評価を行い、第 6 章で今後の課題と結論を述べる。

2. 関連研究

タスクによる並列プログラミングを行うことで既存のデータ並列化では処理が難しかった不規則なワークロードの並列化が可能になるだけでなく、スレッド間の同期を細粒度化し、同期コストを減らすことができる [1][2][3]。渡部ら [4] は OpenMP のタスクプログラミングモデルを用いて FPGA 上で DGEMM カーネルを計算するときの回路の数とデータのサイズのトレードオフについて研究を行った。OpenMP タスク並列化による Batched Kernel API 生成のアイディアはこの論文から影響を受けたものである。

```
1  void calc_dgemms(const int num_kernels,
2                   const int n,
3                   double **A, double **B,
4                   double **C, double *DMONE,
5                   double *DONE) {
6    for (int i = 0; i < num_kernels; i++) {
7      cublasDgemm(handle, CUBLAS_OP_N,
8                  CUBLAS_OP_N, n, n, n,
9                  &DMONE, A[i], n, B[i], n,
10                 &DONE, C[i], n);
11     cudaDeviceSynchronize();
12   }
13 }
```

図 1 ループ文を用いた DGEMM カーネルの計算

現在利用可能な Batched Kernel API としては Batched BLAS が挙げられる。Intel Math Kernel Library (MKL)[5] は Intel CPU アーキテクチャ向け Batched BLAS を実装している。NVIDIA CUDA プログラミング環境も cuBLAS[6] と呼ばれる NVIDIA GPU アーキテクチャ向け BLAS 実装を提供している。cuBLAS も Batched DGEMM などの Batched BLAS API を提供する。現状の Batched BLAS のもっとも大きい問題点は標準仕様が存在しないことである。Relton ら [7] は標準仕様の整備を行うため、論文の中でいくつかの Batched BLAS の仕様を比較している。Dongarra ら [8][9][10] は小行列に分割可能な線形問題を解くための Batched BLAS インターフェイスを提案している。提案されたインターフェイスは汎用 CPU やメニーコアアーキテクチャをサポートするすべての BLAS API をカバーするように設計されたものである。これらのインターフェイスは Batched BLAS の標準仕様を定めるように提案されたものであるが、現状では異なるハードウエアで異なる API が用いられている。スレッド並列化における OpenMP のような抽象度の高いプログラミングモデルが存在しないため、Batched BLAS を用いるには既存の BLAS API からの（手動）コード変換が必要である。

3. Batched Kernel API の概要

本章では Batched Kernel API の概要を述べる。現状利用可能な Batched BLAS を対象 API とするが、本研究で提案するコード変換手法は他の API にも適用可能である。

リスト 1 は NVIDIA GPU 向け cuBLAS を用いる DGEMM 計算コードを示す。コードは複数の DGEMM カーネルをループ文によって計算する。DGEMM カーネルのデータの間に依存性がないと仮定すれば、並列プログラミングモデルを用いることで並列に実行させることができる。図 2 は複数の計算カーネル (赤い丸) を処理するときの GPU の実行モデルを示す。リスト 1 のような逐次コードを実行すると各計算カーネルが順番に実行され、GPU の計算資源を独占する (図 2 の 1)。

OpenMP などの並列プログラミングモデルを使うこと

ⓒ 2019 Information Processing Society of Japan

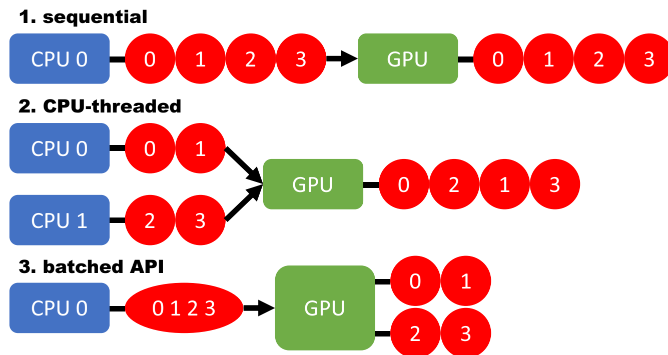

図 2　GPU の実行モデル

```
1  void
2  calc_dgemms_batched(const int num_kernels,
3                      const int n, double **A,
4                      double **B, double **C,
5                      double *DMONE,
6                      double *DONE) {
7    cublasDgemmBatched(handle, CUBLAS_OP_N,
8                       CUBLAS_OP_N, n, n, n,
9                       &DMONE, A, n, B, n,
10                      &DONE, C, n,
11                      num_kernels);
12   cudaDeviceSynchronize();
13 }
```

図 3　Batched BLAS API による DGEMM カーネルの計算

でリスト 1 のコードを並列化することができる。ユーザがループ文に *parallel for* 指示文を挿入することで計算カーネルを CPU の複数の計算コアに割り当て、並列実行させる (図 2 の 2). この方法では複数の CPU スレッドが同じ GPU の実行ストリームを共有しているため、DGEMM カーネルは GPU の中で逐次に実行される[*1]。そのため、常に一つの DGEMM カーネルが GPU の計算資源を独占し、小さいデータサイズを計算するときは一部の計算コアが利用されない。Intel MKL などの CPU 向け BLAS を使うときは CPU スレッドによって DGEMM カーネルが異なる計算コアで計算されるため、コア数に比例して性能が向上する。

Batched Kernel API を使うことで独立した計算カーネルを並列実行させることができる。リスト 3 に cuBLAS の Batched DGEMM の例を示す。従来の *cublasDgemm()* と Batched DGEMM (*cublasDgemmBatched()*) の違いは、Batched DGEMM は計算するデータ (複数) のポインターの配列を引数として受け取ることである。配列を用いてまとめられた複数の DGEMM カーネルをバッチと呼ぶ。cuBLAS の Batched DGEMM はバッチの中の計算カーネルを GPU の計算コアに割り当てることで並列実行を行い、高い性能を達成する (図 2 の 3) [*2]。

[*1] OpenMP の *target* 指示文を使うことで対象コード領域を GPU にオフロードすることが考えられるが、cuBLAS は CPU 側で実行される必要があるため、*target* 指示文を使うことはできない。
[*2] 複数の GPU の実行ストリームを使うように並列化を行うことで Batched BLAS が行うことと同等のことを記述することも可能である。このためには CUDA と OpenMP を組み合わせたコード変換作業と GPU のアーキテクチャや実行モデルに関する専門的な知識が必要である。

```
1  void
2  cholesky_decomposition(const int tile_size,
3                         const int num_tiles,
4                         double* A[nt][nt]) {
5  #pragma omp parallel
6  #pragma omp single
7    for (int k = 0; k < num_tiles; k++) {
8  #pragma omp task depend(out:A[k][k])
9      dpotrf(A[k][k], tile_size, tile_size);
10     for (int i = k+1; i < num_tiles; i++) {
11 #pragma omp task depend(in:A[k][k]) \\
12                  depend(out:A[k][i])
13       dtrsm(A[k][k], A[k][i],
14             tile_size, tile_size);
15     }
16     for (int i = k+1; i < num_tiles; i++) {
17       for (int j = k+1; j < i; j++) {
18 #pragma omp task \\
19             depend(in:A[k][i],A[k][j]) \\
20             depend(out:A[j][i])
21         dgemm(A[k][i], A[k][j], A[j][i],
22               tile_size, tile_size);
23       }
24 #pragma omp task depend(in:A[k][i]) \\
25                  depend(out:A[i][i])
26       dsyrk(A[k][i], A[i][i],
27             tile_size, tile_size);
28     }
29   }
30 }
```

図 4　OpenMP タスクによって並列化された Blocked Cholesky Decomposition

Batched BLAS を使うデメリットとしては、バッチ化できるのは同じ種類のカーネルだけであるため、異なる計算カーネルを含む並列性を活用できないことや、従来の BLAS API からのコード変換が必要であることが挙げられる。本研究の目的は Batched BLAS を含む Batched Kernel API をより抽象的な方法で利用できるプログラミングモデルを提案することである。

4. Batched BLAS API を用いる OpenMP タスク生成

本章では OpenMP の *task* 指示文から Batched Kernel API を生成するコード変換手法を提案する。想定するプログラミング環境ではユーザが BLAS ライブラリを C 言語で記述し、OpenMP 指示文で並列化を行う。対象ハードウェアは独自の計算環境を持つ GPU アクセラレータを考える。このようなタイプのアクセラレータは CPU (ホスト) に接続されており、CPU 側のスレッド並列化がアクセラレータの性能を直接向上させることはない。

4.1 プログラミングモデル

リスト 4 に OpenMP で並列化された Blocked Cholesky

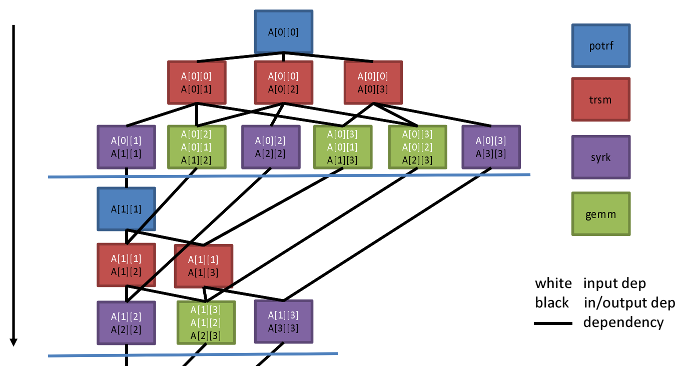

図 5　Blocked Cholesky Decomposition のタスクグラフ

Decomposition コードを示す。このコードは cuBLAS の API 関数を OpenMP の *task* 指示文でタスク化することによって並列性を記述している。リスト 1 で示されているようにアプリケーションの並列性が明確でわかりやすいときは従来のデータ並列化によって並列性を記述することができる。そのようなときはリスト 3 のように Batched Kernel API にコード変換を行うことも比較的簡単である。

しかし、並列性が自明ではないアプリケーションだとそのような手法が適用できない。リスト 4 で示された Cholesky Decomposition は依存関係がある BLAS 計算カーネルがループ文の中で実行されており、イテレーション毎に並列性が不規則である。このようなアプリケーションは動的に生成されるタスクによって並列性を記述することができる。各 BLAS (potrf は LAPACK API 関数) 計算カーネルを *task* 指示文で囲むことで計算カーネルをタスク化する。タスク間の依存関係は *depend* 節で記述する。その様子を図 5 に示す。OpenMP ランタイムによってタスクが動的に生成され、スレッドに割り当てられる。異なるスレッドに割り当てることによって並列実行を実現する。図 5 の同じ横列に並んだタスクは互いに依存性がなく、並列に実行することができる。また依存関係が解消されたタスク (入力エッジがあるタスクが実行済み) も他のタスクと並列に実行可能になる。このような動的な並列性は一見自明ではなく、ユーザが並列性を見つけ出して手動でバッチ化することは難しい。

本研究では OpenMP の処理系が事前に登録された数値計算ライブラリの計算カーネルを検知してバッチ化を行うコード変換手法を提案する。バッチ化を行うためには処理系によるコード変換とランタイムライブラリによるバッチタスクの生成とスケジュールが必要である。次の節でそれぞれについて説明を行う。

4.2　コード変換

著者らは Omni Compiler Infrastructure[11] (Omni コンパイラ) を用いて OpenMP 処理系の開発を行っている。提案手法の *task* 指示文によるコード変換も Omni コンパイラ

```
1  void cholesky_decomposition(const int
2        tile_size, const int num_tiles,
3        double* A[nt][nt]) {
4    for (int k = 0; k < nt; k++) {
5      void *args[3]
6        = {A[k][k], &tile_size, &tile_size};
7      void *out_data[1] = {A[k][k]};
8      task_create(0, potrf, 3, args, 0,
9                  NULL, 1, out_data);
10     for (int i = k + 1; i < nt; i++) {
11       void *args[4]
12         = {A[k][k], A[k][i],
13            &tile_size, &tile_size};
14       void *in_data[1] = {A[k][k]};
15       void *out_data[1] = {A[k][i]};
16       task_create(0, trsm, 4, args,
17                   1, in_data, 1, out_data);
18     }
19     for (int i = k + 1; i < nt; i++) {
20       for (int j = k + 1; j < i; j++) {
21         void *args[5]
22           = {A[k][i], A[k][j], A[j][i],
23              &tile_size, &tile_size};
24         void *in_data[2]
25           = {A[k][i], A[k][j]};
26         void *out_data[1] = {A[j][i]};
27         task_create(1, gemm, 5, args,
28                     2, in_data,
29                     1, out_data);
30       }
31       void *args[4]
32         = {A[k][i], A[i][i],
33            &tile_size, &tile_size};
34       void *in_data[1] = {A[k][i]};
35       void *out_data[1] = {A[i][i]};
36       task_create(0, syrk, 4, args,
37                   1, in_data, 1, out_data);
38     }
39   }
40   task_waitall();
41 }
```

図 6　変換後の Blocked Cholesky Decomposition コード

を用いて実装を行う。Omni コンパイラは source-to-source のコード変換を行うため、OpenMP 指示文とベース言語 (本研究では C 言語を想定) で記述された逐次コードは OpenMP ランライム関数呼び出しを含むベース言語のコードに変換される。リスト 4 から変換されたコードをリスト 6 に示す[*3]。処理系は *task* 指示文から動的タスク生成を行うランタイム関数 *task_create()* を生成する。対象コード領域は関数化され、実行に必要なデータが引数として与えられる。*depend* 節が記述されたときはその値が *task_create()* の引数として与えられ、依存性の解析とタスクスケジュールに使われる。

対象コード領域が事前に登録された数値計算ライブラリの関数を呼び出している場合、バッチ化の対象になる。プロトタイプの処理系は対象コード領域が BLAS の DGEMM 関数を呼び出しているとき、それを検知してバッチ化の候

[*3] 変換後のコードは理解しやすくするために簡略されている。

ⓒ 2019 Information Processing Society of Japan

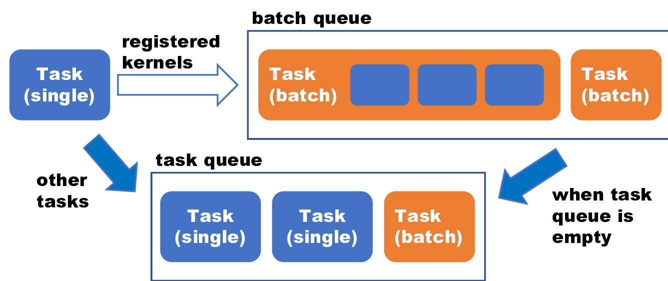

図 7　バッチ化されたタスクのスケジュール

補に分類する。バッチ化の対象になったタスクはユニークな数値 (バッチ ID) が与えられる。バッチ ID は呼ばれる BLAS 関数の種類や引数の値によって異なる値を持つようになる。DGEMM の場合、異なる行列であっても、同じサイズのものを処理する関数呼び出しには同じバッチ ID が与えられる[*4]。*task_create()* は最初の引数としてバッチ ID を受け取る。生成されるタスクがバッチ化の対象でない場合、バッチ ID は 0 である。

4.3　ランタイムライブラリの実装

バッチ ID が 0 以上の値を持つタスクはランタイムによってバッチ化される。図 7 にバッチを扱うために変更された OpenMP のタスクランタイムの動作を示す。OpenMP のタスクランタイムは依存関係が解消され実行可能になったタスクを格納するタスクキューを持つ。CPU などの計算資源が利用可能になったときに次の実行されるタスクをタスクキューから取り出して計算資源に割り当てることで並列実行を行う。次に実行されるタスクを選択する方法は実装依存であり、各 OpenMP ランタイムは独自のスケジュールアルゴリズムを持つ。提案手法のために新しいキューをランタイムに追加する。図 7 に示したようにバッチキューと呼ばれる新しいキューがバッチ化された DGEMM カーネルを操作するために追加されている。

入力関係にあるタスクが実行されて依存性が解消される前は、ランタイムはバッチ化の対象であるかどうかに関わらず同じ動きをする。依存関係が解消されて実行可能になったタスクはスケジュールの対象になる。バッチ化の対象ではない (DGEMM を含まない) タスクは通常通りタスクキューに入れられる。

バッチ ID が与えられてバッチ化の対象になったタスクはバッチキューに入れられる。タスクがバッチキューの中でそのバッチ ID を持つ最初のものであれば空のバッチタスクを生成し、タスクの中身をコピーする。バッチキューの中に既に同じバッチ ID を持つタスクが存在したら、ランタイムは二つのタスクをマージしてバッチを生成する。バッチの生成手法は対象 API 毎に異なる。実装者はそのための情報を処理系に登録しておかなければならない。Batched DGEMM の場合はデータサイズが同じ異なる行列をバッチ化することができるため、行列 A、B、C のポインターがポインター配列に格納される。呼び出す関数 API も従来の DGEMM から Batched DGEMM に置き換えられる。

バッチタスクは OpenMP タスクの構造体を継承するため、OpenMP のタスクとして扱うことができる。したがって、利用可能な計算資源に割り当てることで並列実行を行うことも可能である。実装したランタイムでは通常のタスクキューにタスクが残っているときはバッチキューのタスクを実行しない。タスクキューのタスクがすべて実行され、空になったときにバッチキューからタスクキューにタスクを動かしてスケジュール可能な状態にする。これはバッチキューにあるタスクの実行を遅延させることで、できるだけ多くの API 関数がバッチにマージされるようにするためである。このような手法を使うことで図 5 の横列に並ぶ DGEMM タスクがすべて同じバッチにマージされて実行できるようになる。

OpenMP タスクを実装するために Argonne National Laboratory が開発を行っている Argobots[12] スレッドライブラリを用いる。Argobots はユーザレベルで実装された軽量スレッドを生成、管理するための API を提供する。実装されたランタイムライブラリはバッチ化のために専用のスレッドを一つ生成する。単一スレッドを用いる理由は図 2 で示したように複数のホストスレッドが GPU の性能向上に貢献しないと考えたことや、第 5 章の性能解析を簡単にするためである。

*4　このためには処理系によるコード解析が必要であるが、プロトタイプの処理系ではすべての DGEMM タスクに同じバッチ ID を与えている

表 1　評価環境

Item	Name/Value
CPU	Intel (R) Xeon (R) CPU E5-2680 8 cores (×2 sockets), 2.70 GHz
Memory	DDR4 64 GB
GPU	NVIDIA Tesla K20
Back-end Compiler	Intel Compiler 18.0.3
CUDA Library	CUDA 9.1 with cuBLAS

5.　性能評価

本章では実装したバッチ化 OpenMP 処理系の性能評価を行う。ベンチマークとして複数の DGEMM 計算カーネルをループ文で処理する DGEMM ループとリスト 4 で示した Blocked Cholesky Decomposition を用いる。表 5 に評価環境を示す。GPU アクセラレータは NVIDIA Tesla K20 GPU を、BLAS 実装には CUDA 9.1 の cuBLAS の Batched BLAS を用いた。

5.1　DGEMM ループ

リスト 8 に DGEMM ループのコードを示す。ループ文の各イテレーションで独立した DGEMM 計算カーネルを

ⓒ 2019 Information Processing Society of Japan

```
1  void dgemm_loop(const int num_kernels,
2                  const int n, double **A,
3                  double **B, double **C) {
4  #pragma omp parallel
5  #pragma omp single
6    for (int k = 0; k < num_kernels; k++) {
7  #pragma omp task
8      dgemm(A[k], B[k], C[k], n, n);
9    }
10 }
```

図 8 OpenMP タスクによる複数 DGEMM カーネルの計算

cuBLAS API を使って計算するものである。簡単なベンチマークコードであり、高い並列性を持つため、処理系の基本性能を計ることに役立つ。

図 9 に DGEMM ループの性能を示す。*ser* は逐次コード、*thread* は OpenMP の task 並列化版をホスト側のマルチコア CPU で実行したとき (図 2 の 2、通常の OpenMP ランタイムを利用) の性能である。評価のために CPU の計算コアと同数の 16 個の OS スレッド (Argobots の ES) を生成した。*batch* は本研究で提案を行ったバッチ化を行う OpenMP 処理系とランタイムによる評価結果を示す。バッチ化は専用スレッドで行われるため、バッチタスクの生成と管理、実行は一つの OS スレッドで実行される。DGEMM ループの場合、すべての DGEMM カーネルを単一バッチにまとめられる (バッチタスクの生成が延期されるため)。評価結果は複数 DGEMM の逐次実行と Batched DGEMM の性能比較と考えることができる。

行列のサイズが小さい (16×16、32×32) ときは一つの DGEMM カーネルが GPU の計算資源を独占するため、逐次実行の効率が悪い。そのため、DGEMM カーネルの数を増やしていくとバッチ化と GPU 上での並列実行によって性能が向上する。行列のサイズを増やしていくと一つの DGEMM カーネルでも GPU の複数の計算資源を活用できるようになり、逐次実行とバッチ化による性能が近づく。*thread* の場合は並列実行はホスト側の CPU のみであり、DGEMM カーネル実行は逐次化される。しかし、ホスト側も逐次実行を行う *ser* と比べると性能が向上していることがわかる。これは DGEMM カーネルの内部処理や OpenMP ランタイムの処理がマルチスレッド化されるため、並列化のオーバーヘッドや同期コストが削減されるためである。

5.2 Blocked Cholesky Decomposition

データ並列化でも並列化できる DGEMM ループと違って、Blocked Cholesky Decomposition は不規則な依存関係と並列性を持つ。そのため、OpenMP タスクプログラミングモデルを用いるバッチ化の実践的な例として考えることができる。評価のため、リスト 4 に示したコードを利用した。Blocked Cholesky Decomposition では対象の行列を小行列 (tile) に分割して計算を行う。ソースコードの *tile_size* は小行列のサイズをあらわし、入力パラメータとしてユーザが指定することができる。ベンチマークでもっとも多く実行されるのは二つの小行列の行列積を計算する DGEMM カーネルである。したがって、DGEMM のバッチ化によって性能が改善すると予想される。

図 10 に Blocked Cholesky Decomposition の性能を示す。*ser*、*thread*、*batch* の意味は図 9 と同じである。DGEMM カーネルの数を増やして性能の変化を見るため、小行列のサイズを固定して対象行列のサイズを増加させる (小行列の数が増加)。小行列のサイズが小さいときはいくつかのケースでバッチ化による性能改善がみられるものの、大きい問題サイズでは逐次実行より性能が低い。より大きい小行列を用いることで DGEMM カーネルの問題サイズを大きくするとバッチ化による性能改善がよりはっきり表れるようになる。

図 11 に Blocked Cholesky Decomposition の評価結果の詳細を示す[*5]。*dgemm* は従来の DGEMM の実行時間の割合、*dgemm batch* は Batched DGEMM の実行時間の割合を示す。*math* は DGEMM 以外の BLAS カーネルと LAPACK の potrf カーネルの割合の合計を示す。*system* は残りの実行時間の割合であり、主にタスクの管理を含む OpenMP ランタイムのオーバーヘッドである。

4096-32 は行列サイズが 4096、小行列のサイズが 32 の時の性能詳細で、図 10 の中でもっとも DGEMM カーネルの実行が多いケースである。バッチ化によって DGEMM カーネルが GPU 上で並列実行されるようになり、実行時間の合計は 11.8 秒から 0.27 秒に改善される。OpenMP ランタイムのタスク管理のオーバーヘッドは生成されるタスクの数に比例する。小さい行列を計算するときは DGEMM カーネルの実行時間がランタイムのオーバーヘッドより小さくなることから、全体の性能効率が低下する。

4096-64 と *4096-128* は問題の行列サイズを 4096 に固定し、小行列のサイズをサイズを大きく (64、128) した結果である。その結果、分割の数が減ることから DGEMM カーネルの呼び出し頻度は下がり、1 回の計算が大きくなる。タスクの数が減少し、DGEMM の計算時間が長くなることによって OpenMP ランタイムのオーバーヘッドが相対的に小さくなり、実行効率が向上するように見える。DGEMM ループでもわかるように本来小さい DGEMM カーネルの数が多くなるほどバッチ化の効果が大きい。問題のサイズを固定した場合、小行列のサイズを増やすことによる性能のトレードオフ (DGEMM タスクの数と対象小行列の大きさ) があることがわかる。今回の評価結果では *4096-128* のケースでもっとも高い性能改善が見られた。

[*5] マルチスレッド化されたアプリケーションの性能詳細をわかりやすく示すことが難しい、またそれが本研究のトピックから外れるため、*thread* の性能詳細は示していない。

ⓒ 2019 Information Processing Society of Japan

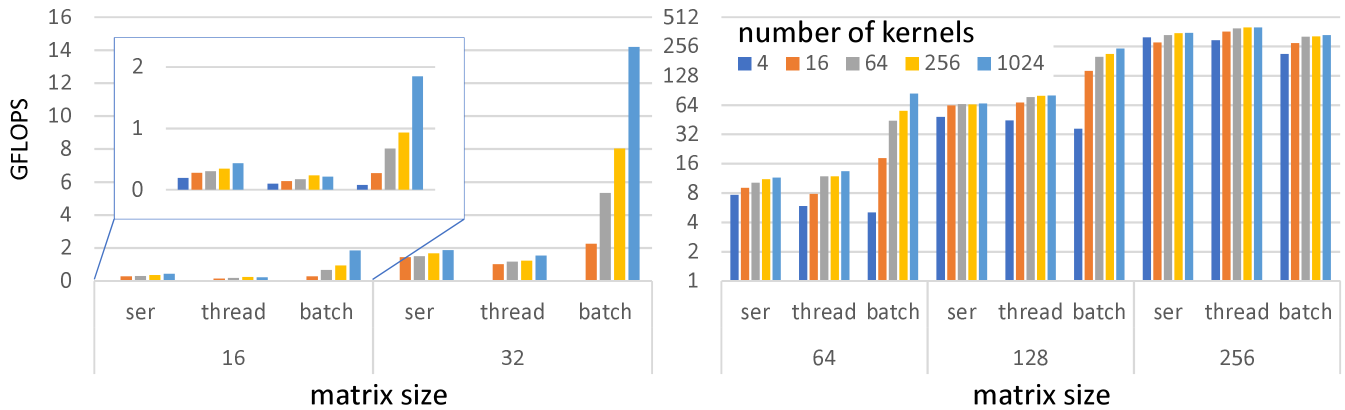

図 9 DGEMM ループの性能

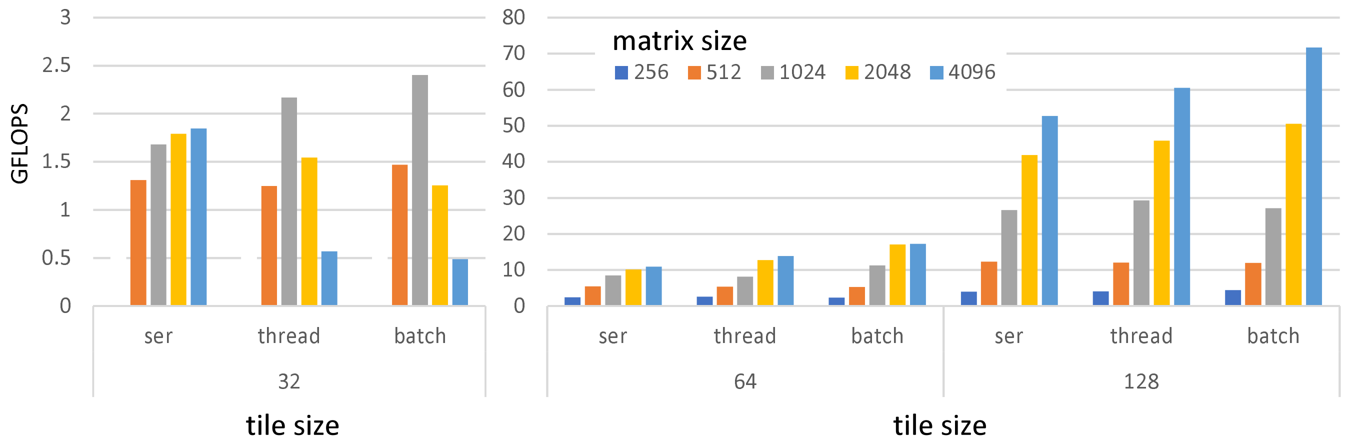

図 10 Blocked Cholesky Decomposition の性能

性能評価の結果、提案手法により DGEMM カーネルの計算性能が向上することが確認できた。しかし、ランタイムライブラリの中のバッチタスクのオーバーヘッドが存在することから、データサイズが小さすぎる場合は性能が低下することが見られる。この問題を解決するためにランタイムライブラリのマルチスレッド化を行い、バッチタスクを並列に生成、管理することで最適化を行う。

6. おわりに

本研究では OpenMP プログラミングモデルを用いて Batched Kernel API を生成するコード変換手法の提案を行った。NVIDIA GPU を用いた性能評価では充分な数のタスクが存在する場合、処理系によるバッチ化によって性能が向上することが確認できた。プロトタイプの処理系は GPU と BLAS DGEMM API を対象にしているが、提案手法は汎用 CPU や他の Batched Kernel API を提供する数値計算ライブラリに適用可能なものである。

今後の課題として以下のようなものが挙げられる。

- レベル 3 の BLAS API をすべてサポート
- バッチタスク処理のオーバーヘッドを減らすためのランタイムライブラリの最適化
- GPU 上で異なる計算カーネルを並列実行するためのマルチストリーム化
- マルチ GPU、CPU と GPU の並列実行のサポート

参考文献

[1] Jin, C. and Baskaran, M.: Analysis of Explicit vs. Implicit Tasking in OpenMP Using Kripke, pp. 62–70 (online), DOI: 10.1109/ESPM2.2018.00012 (2018).

[2] Olivier, S. L. and Prins, J. F.: Evaluating OpenMP 3.0 Run Time Systems on Unbalanced Task Graphs, *Proceedings of the 5th International Workshop on OpenMP: Evolving OpenMP in an Age of Extreme Parallelism*, IWOMP '09, Berlin, Heidelberg, Springer-Verlag, pp. 63–78 (online), available from ⟨"http://dx.doi.org/10.1007/978-3-642-02303-3_6"⟩ (2009).

[3] Muddukrishna, A., Jonsson, P. A., Vlassov, V. and Brorsson, M.: *OpenMP in the Era of Low Power Devices and Accelerators: 9th International Workshop on OpenMP, IWOMP 2013, Canberra, ACT, Australia, September 16-18, 2013. Proceedings*, chapter Locality-Aware Task Scheduling and Data Distribution on NUMA Systems, pp. 156–170 (online), DOI: 10.1007/978-3-642-40698-0_12, Springer Berlin Heidelberg (2013).

[4] Watanabe, Y., Lee, J., Boku, T. and Sato, M.: Trade-Off of Offloading to FPGA in OpenMP Task-Based Programming, *Evolving OpenMP for Evolving Architectures* (de Supinski, B. R., Valero-Lara, P., Martorell, X., Mateo Bellido, S. and Labarta, J., eds.), Cham, Springer International Publishing, pp. 96–110 (2018).

ⓒ 2019 Information Processing Society of Japan

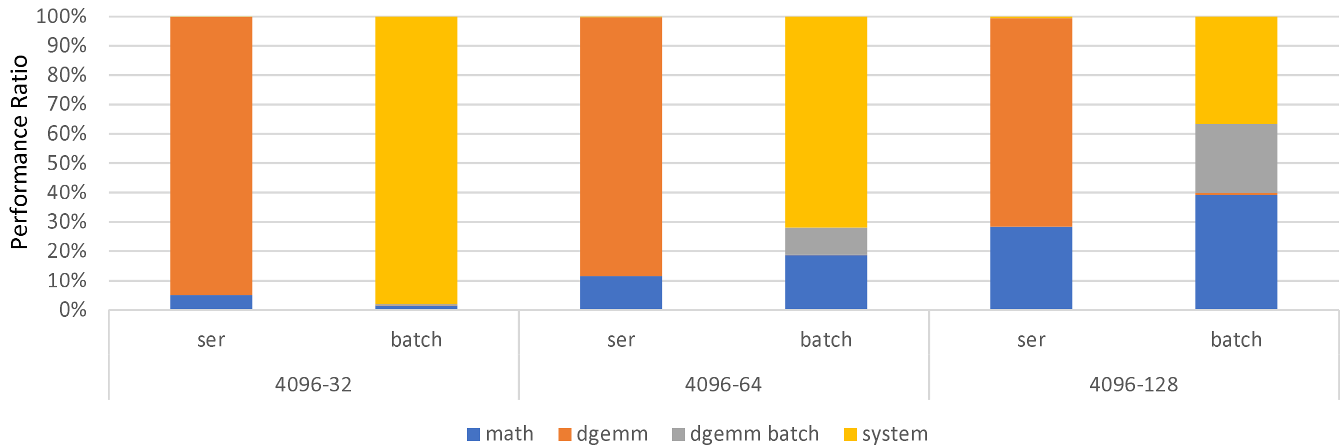

図 **11** 実行性能の詳細

[5] Intel Math Kernel Library - Batched DGEMM Interface: https://software.intel.com/en-us/mkl-developer-reference-c-cblas-gemm-batch.

[6] NVIDIA cuBLAS - Batched DGEMM Interface: https://docs.nvidia.com/cuda/cublas/index.html#cublas-lt-t-gt-gemmbatched.

[7] Relton, S. D., Valero-Lara, P. and Zounon, M.: A Comparison of Potential Interfaces for Batched BLAS Computations (2016).

[8] Dongarra, J. J., Duff, I. S., Gates, M., Haidar, A., Hammarling, S., Higham, N. J., Hogg, J., Valero-Lara, P., Relton, S. D., Tomov, S. and Zounon, M.: A Proposed API for Batched Basic Linear Algebra Subprograms (2016).

[9] Dongarra, J., Hammarling, S., Higham, N. J., Relton, S. D., Valero-Lara, P. and Zounon, M.: The Design and Performance of Batched BLAS on Modern High-Performance Computing Systems, *Procedia Computer Science*, Vol. 108, pp. 495 – 504 (online), DOI: https://doi.org/10.1016/j.procs.2017.05.138 (2017).

[10] Dongarra, J., Duff, I., Gates, M., Haidar, A., Hammarling, S., Higham, N. J., Hogg, J., Lara, P. V., Luszczek, P., Zounon, M., Relton, S. D., Tomov, S., Costa, T. and Knepper, S.: Batched BLAS (Basic Linear Algebra Subprograms) 2018 Specification (2018).

[11] Omni Compiler Infrastructure: https://omni-compiler.org/.

[12] Argobots - Official Repository on Github: https://github.com/pmodels/argobots.

ⓒ 2019 Information Processing Society of Japan