Vol.2020-HPC-173 No.11
2020/3/17

演算加速機構を対象とする タスク並列ランタイムにおけるデータ移動最適化

渡部 裕[1,a)]　李 珍泌[2]　朴 泰祐[3,1]　佐藤 三久[2,1]

概要：
演算加速機構を対象とするタスク並列モデルの最適化を実現するランタイムの設計および実装を行った．様々な演算加速機構に対応可能な設計を行い，現時点では NVIDIA GPU および Intel FPGA に対応する．最適化として，ホスト・演算加速機構間のデータ移動をランタイム内で管理し，ユーザによる明示的なデータ移動の指示を用いずとも移動回数を削減可能な機構を設計した．また，複数 GPU を用いる場合の演算加速機構間データ移動の最適化についての調査を行った．ブロックコレスキー分解を例に用いた評価では，FPGA の場合において，ホスト・演算加速機構間の最適化の有無により 1.16̃1.18 倍の性能向上を，GPU の場合において 1.64̃8 倍の性能向上を確認し，ユーザによる明示的な記述を必要としない最適化により生産性を保ちつつもタスク並列実行の最適化が可能であることを示した．複数の GPU を用いた評価においては，より多くの GPU を用いる場合に性能が低下することを確認し，またその要因についての調査を行うことで改善の方向性を示した．

1. 序論

タスク間の並列性を活用した新たな実行モデルである「タスク並列モデル」が提案され，多くの研究によってタスク並列モデルの有効性が確認されている．タスク並列モデルでは従来データ並列モデルのような大域同期を必要とせず柔軟な実行が可能であり，「多数のタスクをいかに高速に処理するか」という点でスループット重視の実行モデルとなっている．

一方，近年の大規模計算機のトレンドは GPU (Graphics Processsing Unit) を代表とする演算加速機構を併用する，アクセラレータ型の計算機が主流の 1 つとなっている．GPU などの演算加速機構は並列処理などに特化することでより高い消費電力当たりの実行性能を実現している．TOP500 の上位 10 個のうち，1 位を含む 6 つのスーパーコンピュータはこのアクセラレータ型であり，単一の種類の演算加速機構を搭載する．また，筑波大学　計算機科学研究センターで運用されている Cygnus[1] スーパーコンピュータのように，複数の演算加速機構を搭載するクラスタ計算機も存在する．Cygnus は NVidia V100 GPU[2] に加え，Intel Stratix 10 FPGA[3] を搭載する．GPU が不得意とする並列性の低い計算や複雑な分岐処理を含む計算に対し FPGA を適用することで，アプリケーションの実行性能をより高めることを目標としている．なお，このような複数の演算加速機構を持つ計算機の可能性は他所でも言及されており，今後のスーパーコンピュータのアーキテクチャの 1 つとなりうる可能性がある [4].

本研究では，タスク並列モデルにおいて演算加速機構を効率的に扱うためのランタイムの提案を目標とする．効率的に扱うための機構の 1 つとして，本原稿ではユーザによる明示的な最適化を必要としない，ホスト・演算加速機構間のデータ移動最適化の設計を行った．また，将来的に OpenMP タスク並列モデル，および XcalableMP タスク並列モデルのバックエンドランタイムとして使用することで，複数の演算加速機構を簡便な記述で扱うことを目標とする．

以降，本原稿は次のように構成される．2 章においてタスク並列と演算加速機構について述べる．3 章では提案するランタイムの設計について述べ，4 章で性能評価を行う．5 章では関連する研究を紹介し，6 章で結論と今後の課題を述べる．

2. タスク並列モデルと演算加速機構

本章では，タスク並列モデルにおいて演算加速機構を用いることについて述べる．なお，本研究に至る過程おいて FPGA を対象とするタスク並列モデルについての研究を行い，IWOMP'18 において報告を行っている [5].

2.1 OpenMP タスク並列モデル

タスク並列モデルとは，計算を複数のタスクに分割し各タスクを並列に処理する並列実行モデルである．OpenMP

[1] 筑波大学 システム情報工学研究科
[2] 理化学研究所 計算科学研究センター
[3] 筑波大学 計算科学研究センター
[a)] ywatanabe@hpcs.cs.tsukuba.ac.jp

プログラム 1: OpenMP task プログラムの例

```
#pragma omp parallel
{
#pragma omp single
    {
        for(int i=0; i<N; i++) {
#pragma omp task depend(out:A[i])
            taskA(A[i]);
#pragma omp task depend(out:B[i])
            taskB(B[i]);
#pragma omp task depend(in:A[i],B[i]) depend(out:C[i])
            taskC(A[i], B[i], C[i])
#pragma omp task depend(in:A[i]) depend(in:C[i])
            taskD(D[i]);
        }
    }
#pragma omp taskwait
}
```

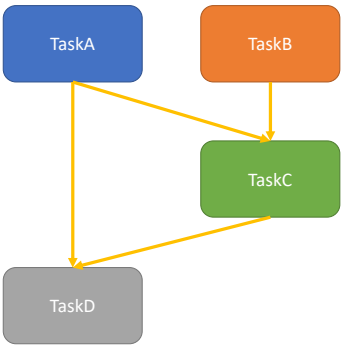

図 1: データ依存グラフ

3.5 で導入され，4.0 でタスク間の依存性の記述が導入されたことでより多くのアプリケーションをタスク並列で記述可能となり，大きな注目を集めるようになった．また，大域同期を必要としないことや，処理のオーバーラップを容易に記述可能である点も注目を集めている点も注目を集める要因である．

OpenMP タスク指示文を用いたプログラムの例をプログラム 1 に示す．OpenMP では task 指示文を用いてタスクを生成し，depend 節を用いてタスク間のデータ依存性を記述する．このプログラムでは task C が taskA と taskB の出力を入力として使用し，taskD は task A および taskC の出力を入力として用いることを仮定している．そのため，task A では生成すデータする A[i] を，task B では B[i] を out 依存性として depend 節に記述する．タスク並列プログラムを実行する際に動的にタスクが生成され，依存性に基づきタスクフロー図が生成される．この場合に生成されるタスクフロー図は図 1 の通りである．タスクフロー図が生成されたのちにタスクの処理に入るが，この場合，依存性のために taskA，taskB のみが実行可能であるため，それぞれを初めに処理する．その後 taskC が実行可能となる．taskC が終了すると taskD が実行され，taskD の終了とともにタスク並列処理が終了する．

2.2 タスク並列モデルを用いることのメリット

すでに述べた通り，タスク並列モデルはタスク間の依存性に基づき並列処理するモデルであり，いかに多数のタスクを高速に処理するかという「スループット重視」の実行モデルである．したがって，CPU よりも高い演算性能を持つ演算性能加速機構に一部のタスクをオフロードすることにより，さらに高いスループットを実現可能であると考えられる．GPU などの演算加速機構もスループット重視のアーキテクチャであることから親和性も良いと考えられる．

また，処理と処理のオーバーラップを用意に記述可能であることから，CPU 上のタスクと演算性能加速機構上のタスクのオーバーラップを容易に記述可能である点もメリットといえる．タスク並列モデルを用いない場合，計算のオーバーラップはユーザによって記述される必要があったが，タスク並列モデルにおいではデータ依存性のみを記述することでオーバーラップを行うことが可能である．

また OpenMP の場合，target 指示文を用いて演算機構上のタスクを生成することが可能である．タスクごとに違う種類の演算加速機構にオフロードするような記述を行うことで，従来のように複数の演算加速機構ランタイムをユーザの責任で組み合わせて使用するといった複雑さを解消可能であり，可読性を保ったまま複数の演算加速機構を用いることも可能となる．

2.3 演算加速機構をタスク並列モデルで使用するための課題

演算加速機構をタスク並列モデルで使用するための課題の 1 つはホスト・演算加速機構間のデータ移動である．タスク並列では，タスク間の依存性を示す「タスク依存グラフ」が動的に生成され，またタスクの処理も動的に行われることから，ユーザの責任で最適化を行うのは容易ではない．一方，特にブロックコレスキー分解のようなブロック化された計算においては，1 つのブロックを再利用するような構造となっていることが多く，ホスト・演算加速機構間のデータ移動の最適化が性能向上への重要な要素となる．そこで本研究では，ユーザによる明示的な最適化を必要としない最適化機構をタスク並列ランタイムで実現することを目標とする．

また，複数の演算加速機構を用いる場合においては演算加速機構間のデータ移動も課題となる．これを最適化するためには演算加速機構間のデータ移動回数を考慮したタスク並列のスケジューリングを行う必要があるが，現時点では実現できていない．そのため，本研究では初期調査としてラウンドロビンを用いたスケジューリングを使用し，複数の GPU を用いる場合において，ラウンドロビン形式にタスクのオフロード先 GPU を設定した上で，GPU 間のデータ移動方法の違いによる性能の変化について調査を行う．1 つ目の手法は GPU の間直接通信を活用し，GPU1 で実行されるタスクに必要なデータが GPU0 上に存在する場合において GPU0 のメモリを直接読み書きする手法であり，2 つめの手法は演算加速機構間コピーを用いて GPU1 にコピーした上で読み書きを行う手法である．

2.4 OpenMP による記述

OpenMP タスク並列を用いて記述する場合のプログラム

プログラム 2: OpenMP を用いる場合の想定される記述例

```
#pragma omp declare variant(void gemm(float *A, float *B, float *C, unsigned int
        SIZE)) match(context={target} device={target_device_type})
extern void target_gemm(float *A, float *B, float *C, unsigned int SIZE);

void cholesky(const int ts, const int nt, float* A[nt][nt])
{
#pragma omp parallel
#pragma omp single
  for (int k = 0; k < nt; k++) {
#pragma omp task depend(out:A[k][k])
    { potrf(A[k][k], ts, ts); }
    for (int i = k + 1; i < nt; i++) {
#pragma omp task depend(in:A[k][k]) depend(out:A[k][i])
      { trsm(A[k][k], A[k][i], ts, ts); }
    }
    for (int i = k + 1; i < nt; i++) {
      for (int j = k + 1; j < i; j++) {
#pragma omp target depend(in:A[k][i], A[k][j]) depend(inout:A[j][i]) map(in:A[k
        ][i][:]) map(in:A[k][j][:]) map(inout:A[j][i][:]) device(
        target_device_type) async
        { gemm(A[k][i], A[k][j], A[j][i], ts);}
      }
#pragma omp task depend(in:A[k][i]) depend(out:A[i][i])
      { syrk(A[k][i], A[i][i], ts, ts); }
    }
  }
#pragma omp taskwait
}
```

例をプログラム 2 に示す．なお，現時点では OpenMP コンパイラは実装中である．また，現時点ではすでに実装されたカーネル，あるいは CUBLAS などのライブラリを用いることを想定しているが，指示文ベースからのターゲットコード生成についても今後対応を行う必要がある．

task 指示文を用いてホスト上のタスクを，target 指示文を用いて演算加速機構上のタスクを生成する．target 指示文を用いる際，map 節を用いてホスト・演算加速機構のデータのマッピングおよびデータ移動を，device 節を用いてオフロード先の演算加速機構を設定する．また，ここでは target に対し async 指示文の拡張を加えており，演算加速機構のタスクを実行後に終了を待たずにタスクの切り替えを行うことでオーバーラップを実現する．なお，類似した概念として OpenMP にはすでに nowait 指示文が存在しており，nowait 指示文で代用できる可能性もある．

通常，このプログラムの場合は target タスクを実行するたびに map 節に基づきデータ移動が行われることになる．target data update 指示文などを使用してデータ移動を手動で最適化することは可能であるが，一方多数のデータ移動が発生するタスク並列プログラムにおいてデータ移動の最適化は容易ではない．そこで本研究では，このプログラムのままデータ移動をランタイムで最適化する．

なお，FPGA 向け拡張として num_units 節や work_item 節の拡張を検討しており，async 節を含めこれらの詳細は [5] で紹介している．

3. 設計

本章では，演算加速機構を対象とするタスク並列ランタイムの設計について述べる．このランタイムの名称を libompctarget とする．libompctarget では，ホスト上のタスク生成や管理，実行等に加え，演算加速機構へのオフロードやオフロードタスクの管理，ホスト・演算加速機構間のデータ移動最適化等を行う．なお，理化学研究所の佐藤らによる，Argobots を用いたホスト上のタスク並列ランタイムをベースとし設計を行った．

プログラム 3: target タスクの生成

```
1  void *args[2];  void *in_data[1]; void *out_data[1]; void *map_descs[2];
2  args[0] = (void*) A;  args[1] = (void*) B;
3  in_data[0] = (void*) A;
4  out_data[1] = (void*) B;
5  map_descs[0] = make_map_desc(A /*host buffer */, 0 /* offset */, ts*ts*sizeof(
       float) /* size */, 0 /* copy type */);
6  map_descs[1] = make_map_desc(B, 0, ts*ts*sizeof(float), 1);
7
8  kernel = ompc_target_get_kernel("kernel_0");
9  queue_id = ompc_target_create_queue();
10
11 tasklet_target_create(kernel /*kernelobj*/, queue_id /*device queue id*/ , 2 /*
       num of args*/, args, 1 /*num of in deps*/, in_data /*in deps*/, 1 /*num
       of outdeps*/, out_data /*outdeps*/, 2 /*num of map infos*/, map_descs /*
       map infos*/, 1 /*isAsync*/);
12 //tasklet_create(hostFunc, 2, args, 1, in_data, 1, out_data);
```

3.1 libompctarget を用いた target タスク生成例

libompctarget を用いた target タスク生成の例をプログラム 3 に示す．kernel オブジェクトには演算加速機構上のカーネルオブジェクトを格納する．なお，OpenCL の場合は関数名を用いてカーネルオブジェクトを生成する一方，CUDA の場合は関数ポインタとしてカーネルオブジェクトが存在するため，用いる演算加速機構上によってカーネルの生成方法が異なる．また，タスク並列実行に必要なタスクのデータ依存性を in_data, out_data に記述するに加え，ホスト・演算加速機構間のデータのマッピング情報を map_descs に示す．make_map_descs はデータマッピングの情報を pack する関数であり，ホストオブジェクトのポインタ，コピーの開始位置，コピーサイズ，コピーの方向を示す．コピーの方向の例として，0 はホストからデバイスへのコピー，1 はデバイスからホストへのコピーを示す．この情報は target 指示文の map 節から得ることを想定する．最後に，tasklet_target_create 関数により target タスクの生成を行う．このように，API 上では演算加速機構上のメモリの確保などをユーザは行う必要がなく，データ移動最適化を含めすべてランタイム内で行う．

なお，9 行目は CPU 上のタスクを生成する場合に用いられる tasklet_create 関数であり，この場合カーネルオブジェクトではなくホストにあるタスクの関数を引数に設定する．

3.2 演算加速機構の管理を行うランタイム

はじめに複数の種類の演算加速機構に対応するランタイムの設計について述べる．このランタイムの名称を libompctarget とする．

libompctarget では，図 2 に示す通り様々な演算加速機構の演算加速機構ランタイムを抽象化した API の提供を目標とする．本原稿執筆時点では，CUDA および OpenCL API に対応する．なお，評価において NVIDIA GPU を用いるために CUDA を，Intel FPGA を用いるために OpenCL を用いる．また，新たな演算加速機構に対応するためには，その演算加速機構のランタイムを追加することで対応可能な設計とする．

3.3 ホスト・演算加速機構間データ移動の最適化機構

はじめに，ホスト・演算加速機構のオブジェクトの対応を図 3 に示すデータ構造を用いて管理する．チェイン法を

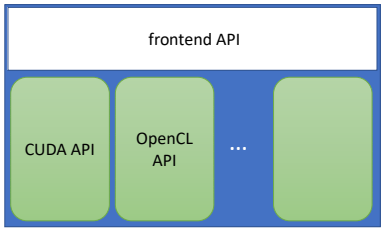

図 2: libompctarget の概要

ベースとしたデータ構造を使用し，ホスト・演算加速機構の対応を 1:N で行う．なお，この N は用いる演算加速機構の数であり，あるホストオブジェクトに対応するオブジェクトは各演算加速機構上に 1 つのみ存在する．この対応表を用いることで，すでに演算加速機構上にメモリが確保されているか，あるいは新たに確保する必要があるかを管理することができる．

また，タスク並列モデルにおいてユーザが記述するデータ依存性を用いることで，最新のデータがどこに存在するか状態を管理する．管理を行う対象のデータは，先述した対応を持つもののみとする．データの更新の検出については，タスク並列モデルにおいてユーザが記述するデータ依存性を活用する．例えば演算加速機構上のタスクが実行された際，out 依存性として記述されたデータが演算加速機構上で更新されたデータであるとする．例えばホスト上でタスクが実行された際，該当タスクの out 依存性に記述されているホスト上のデータ領域に対応する領域が演算加速機構上にも存在する場合，最新のデータはホストにあるとする．

これらの対応表および状態を用いることで，以下のような機構を設けることによりホスト・演算加速機構間のメモリコピーを最小化することが可能である．

3.3.1 ホスト上のタスク実行前後の機構

- 実行前
  (1) in 依存性として記述されているデータについて，ホスト・演算加速機構の領域の対応表を確認する
  (2) 対応がある場合最新のデータの所在を確認する
  (3) 演算加速機構上に最新のデータが存在する場合演算加速機構からホストにコピーし対応の状態を更新する
- 実行後
  (1) out 依存性として記述されているデータについて，ホスト・演算加速機構の領域の対応表を確認する
  (2) 対応が存在する場合，対応の状態を更新し最新のデータをホストにあるとする

3.3.2 演算加速機構上のタスク実行前後の機構

- 実行前
  (1) in 依存性として記述されているデータについて，ホスト・演算加速機構の領域の対応表を確認する
  (2) 対応があった場合，最新のデータがどこにあるかを確認する
  (3) ホスト上に最新のデータが存在する場合，ホストから演算加速機構にコピーを行い，対応の状態を更新する
- 実行後
  (1) out 依存性として記述されているデータについて，ホスト・演算加速機構のメモリ状態を更新し，演算加速機構に最新データがあるとする

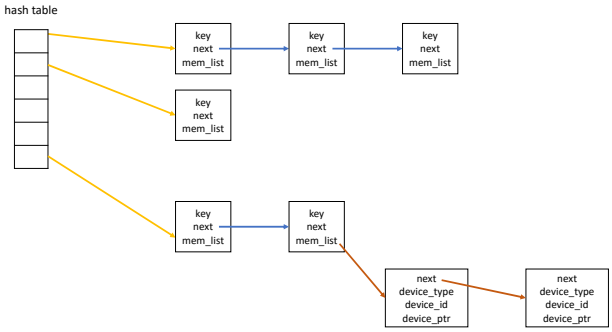

図 3: ホスト・演算加速機構メモリの対応に用いるデータ構造

3.3.3 タスクの実行終了後の機構

(1) ホスト・演算加速機構の対応表を走査する
(2) 走査時に対応の状態を確認し，演算加速機構に最新のデータがある場合のみ演算加速機構からホストにデータをコピーする

4. 評価

本章では，提案するランタイムの評価を行う．ブロックコレスキー分解を例に使用し，FPGA，GPU のそれぞれを用いた場合についての評価を行う．

4.1 ブロックコレスキー分解

ブロックコレスキー分解とは，タスク並列モデルで記述される代表的なプログラムである．ブロックコレスキー分解は以下の 4 つの計算で構成される．

(1) potrf (cholesky factorization)
(2) trsm (triangular matrix equation)
(3) syrk (symmetric rank-k update)
(4) gemm (general matrix multiplication, 行列積)

このうち，ブロック数が増加するごとに行列積タスクの割合が増加し，32×32 ブロックの場合においては全 5984 タスクのうち 4960 ものタスクが行列積となる．したがって，大量の行列積タスクをいかに高速に処理するかが高いスループットを得るための重要な項目となるため，演算加速機構へのオフロード対象とする．

4.2 FPGA を用いた評価

FPGA を用いた評価では，筑波大学計算科学研究センターで運用されている PPX (Pre-PACS Version X) システムの 1 ノードを使用する．評価環境を表 1 に示す．使用する FPGA ボードは Bittware 製の A10PL4[6] であり，本ボードは Arria10 GX FPGA を搭載している．FPGA ボードと Host は PCIe Gen3 x8 により接続される．

次に，実行時システムにおいて最適化しない場合，した

表 1: 評価環境

CPU	Xeon E5-2660 v4 @ 2.00GHz x 2
RAM	DDR4-2400 8GB x 8
FPGA Board	BittWare A10PL4
FPGA	Intel Arria10 FPGA GX115N3F40E2SG
OS	CentOS 7.3 64bit
FPGA Compiler	Intel FPGA SDK for OpenCL 17.1.2.304
Host Compiler	GNU C Compiler 4.8.5

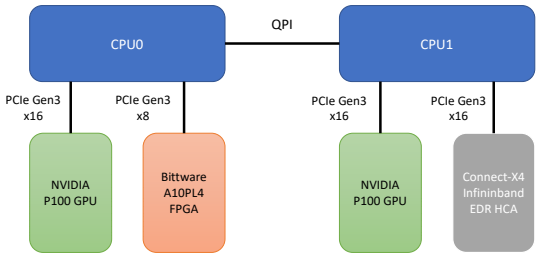

図 4: PPX ノード内のトポロジ

表 2: 問題設定

問題サイズ	32768 x 32768
ブロックサイズ	32 x 32
ブロックあたりのサイズ	1024 x 1024

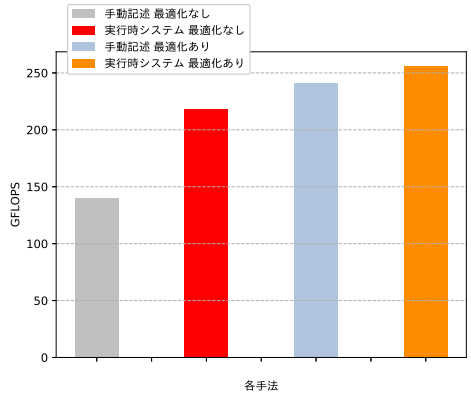

図 5: FPGA でのコレスキー分解を用いた評価

表 3: FPGA を用いた評価におけるホスト・演算加速機構間のメモリコピー回数

memcpy type	w/ opt	w/ o opt
Host to Device	960 (x1)	14880 (x15.5)
Device to Host	465 (x1)	4960 (x10.7)
Total	1425 (x1)	19840 (13.9)

場合の 2 通りについて，マルチコアを用いて実行した場合の性能を 6 に示す．横軸は実行で使用する CPU コア数を示しており，numactl を用いて CPU0 のみを使用するように設定している．結果では，用いる CPU コア数を増やし

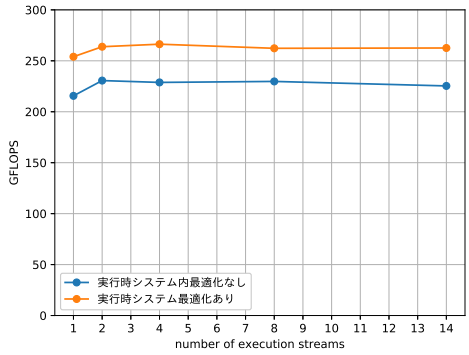

図 6: FPGA でのコレスキー分解を用いた評価 - マルチコア実行

表 4: Node configuration

CPU	Xeon Gold 6126 @ 2.60GHz x 2
Host RAM	DDR4-2600 16GB x 12
InfininBand HCA	Connect-X6 HDR100
OS	CentOS 7.6 64bit
Host Compiler	Intel icc 19.0.3
CUDA version	10.0
User thread library	Argobots 1.0b1

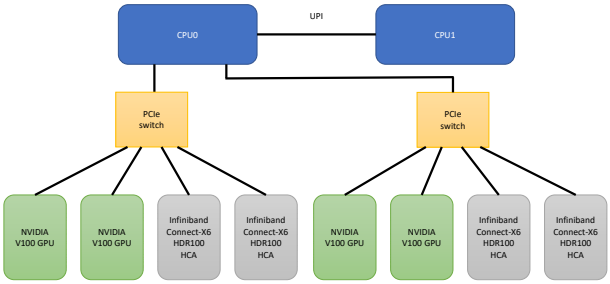

図 7: Cygnus GPU ノード内のトポロジ

た場合でも実行時システム内の最適化により，最適化がない場合と比較し 1 コアの場合において約 1.18 倍，14 コアの場合において約 1.16 倍高い性能が得られている．一方，CPU コアを増やしたとしても全体の性能向上はわずかである．この原因は FPGA での行列積の処理能力が十分ではないためであり，行列積に依存する他の CPU 上のタスクが実行できないためである．FPGA での行列積の処理能力を改善することができれば，使用する CPU コア数を増やした際により高い性能が得られると考えられる．

4.3 GPU を用いた評価

4.3.1 1GPU を用いた評価

初めに 1 GPU を用いた評価を示す．FPGA を用いた評価と同様，ブロックコレスキー分解における行列積タスクを GPU にオフロードする．問題設定についても FPGA 同様，表 2 に従い評価を行う．なお，本評価で用いる GPU は 1 つのみであり，ホスト・演算加速機構間のメモリコピーの

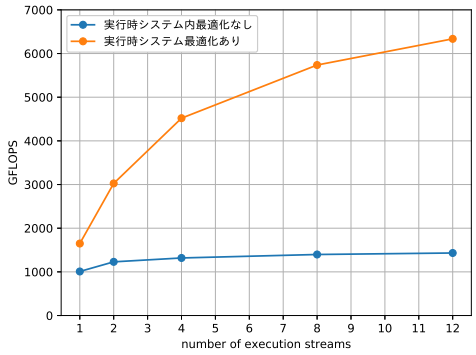

図 8: 1 GPU でのブロックコレスキー分解を用いたホスト・演算加速機構間データ移動最適化に関する評価

最適化による効果の評価が主となる．

図 8 に 1GPU を用いたブロックコレスキー分解の結果を示す．縦軸は性能を示しており，横軸は実行で用いる CPU のコア数を示す．実行においては numactl コマンドを使用し，CPU0 のみを使用するように設定している．

結果が示す通り，ompctarget runtime 内におけるホスト・演算加速機構間メモリコピーの自動最適化の有無により大きな差が出ている．最適化がある場合，ない場合と比較し約 1.6 倍（1 コアの場合） 4.8 倍 (12 コアの場合) 高い性能が得られていることが確認できる．最適化した場合において使用する CPU コア数を増やすごとにより高い性能が得られている理由としては，より多くの CPU コアを使用することでより多くの行列積を一定時間においてオフロードすることが可能となり，結果として GPU の性能をより活用することができたためである．一方，最適化をしない場合においてはメモリコピーのコストのため一定時間あたりの行列積のオフロード回数を改善することができず，結果として CPU コア数を増やしたとしても性能向上にほとんど寄与しなかったと考えられる．なお，メモリコピーの回数については表 3 と同様である．

4.3.2 複数 GPU を用いた評価

実行結果を表 9 に示す．これまで同様，表 2 に従い評価を行っている．この評価では複数の GPU を用いた場合におけるメモリコピーに関する評価を行う．1 つ目の手法は，タスクがオフロードされた GPU とは別の GPU に必要なオブジェクトが存在する場合において直接読み書きを行う手法である．2 つ目は，同様の状況において，タスクの実行前にデバイス間メモリコピーを行い，タスクを実行する GPU にあるメモリのみを実行時に使用する手法である．なお，本評価においてはホスト・デバイス間のデータ移動最適化を有効にしたもののみを用いる．

初めに，1 つ目の手法の結果に注目する．青で示されるグラフは基準となる 1GPU での結果であり，オレンジ，緑のグラフは手法 1 を用いた場合において GPU を 2 台用いた場合，4 台用いた場合の 2 つである．. 結果から，使用する GPU 数を増やすごとに大幅に性能が下がってしまっていることが確認できる．特に性能差が大きいのは GPU を 1 つ用いる場合と 4 つ用いる場合であり，約 6.1 倍 (CPU 1 コアで実行)̃20 倍 (CPU 12 コアで実行) の低下が確認された．この原因は，GPU 直接通信を用いることでメモリアクセスの速度が PCIe の帯域に律速されていることである．PCIe Gen3 x16 の帯域は約 16GB/s であり，Volta GPU に搭載されている HBM2 メモリの 900 GB/s の広帯域を十分に活用することができない．他の GPU に必要とするオブジェクトが存在しない場合はタスクを実行する GPU 上に新しく領域を確保するが，メモリアクセスのロードバランシングが十分にできていないことも考えられる．

次に，2 つ目の手法の結果に注目する．赤，紫で示されるグラフが手法 2 を用いた場合において GPU を 2 台用いた場合，4 台用いた場合の結果である．結果から確認できる通り，手法 1 よりも大幅に性能が改善されている一方，依然として使用する GPU を増やすごとに性能が低下している．手法 1 より高い性能を得られている理由としては，PCIe 間の通信が削減されたこと，タスク実行前にメモリコピーを行いタスク実行中に自身の GPU のメモリのみを使用することで GPU のメモリ帯域をより効率的に使用できたことが考えられる．性能低下については，より多くの GPU を用いることによる CUDA API のオーバーヘッド増加や，ラウンドロビンを用いたオフローディングスケジューリングが要因と考えられる．特にスケジューリングに関し，あるタスクが実行される際に必要なオフジェクトがどこにあるかを考慮したスケジューリングを行うことによってメモリ移動や GPU 上のメモリ確保のコストを削減可能であるため今後改善を行う必要がある．

また，手法 2 を用いた場合のメモリコピー回数を表 5 に示す．ホストから演算加速機構へのコピーに注目すると，より多くの GPU を用いることでコピー回数が増加していることがわかる．設計においては他の GPU に必要なメモリが存在する場合にデバイス間メモリコピーを行うためこの値は一定となることが期待され，実装上の問題であり改善可能であることを確認している．なお，デバイスからホストへのメモリコピー回数に関しては期待通り GPU 数に依存せず一定である．ただしデバイス間のメモリコピーについては，デバイスが増えるごとに増加していることが確認できる．なお，手法 2 を用使用し，問題サイズを 4 倍の 65536×65536 として実行した場合についても図 10 に示す通り同様の傾向が確認された．

次に，1GPU で実行を行った場合と手法 2 を用いて 4GPU で実行を行った場合の 2 つについて，処理を可視化した図を図 11，図 12 に示す．この場合の問題設定は表 2 に従ったものである．これらの色は，表 6 に示す通りの意味を持つ．縦軸の数値はコア番号を示しており，00 - 11 までが CPU コア，12 からは GPU 番号を示す．図から，1GPU と比較し 4GPU ではデバイス間メモリコピーの呼び出しの操作の増加によって実行が遅延しているとわかる．またデバイス間メモリコピーについては，コピー先 GPU のメモリを確保するコストを含むケースも存在する．したがって，デバイス間メモリコピーの増加，またコピー先 GPU のメ

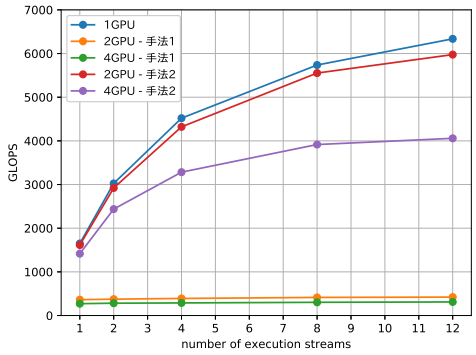

図 9: 複数 GPU でのブロックコレスキー分解を用いた演算加速機構間データ移動最適化に関する評価

モリ確保コストが GPU を増やした際の性能低下の要因の 1 つであると考えられる．

4.3.3 複数 GPU を用いた場合の性能低下に関する調査

データ移動回数の増加やコピー先 GPU のメモリ確保のコスト増加がマルチ GPU 時の性能低下につながっている可能性を確認するため，データ移動のコストと GPU のメモリ確保のコストを削減した場合の性能について計測を行った．問題サイズを 32768 × 32768 とした場合において，データ移動コストを無視した場合の性能，cudaMalloc のコストを無視した場合の性能，およびその 2 つのコストを無視した場合の性能をそれぞれ図 13, 14，15 に示す．問題サイズを 65536 × 65536 とした場合についてもそれぞれ 16, 17，18 に示す．はじめに問題サイズを 32768 × 32768 とした場合に注目する．コストを無視しない場合の性能を示す図 9 とデータ移動コストを無視する場合の図 13 を比較すると，現時点での設計・実装ではデータ移動を十分に隠蔽できておらず性能低下につながっていることが確認できる．また GPU 上のメモリ確保コストを無視した場合の図 14 と比較すると，若干ではあるがより多くの GPU を用いた場合に高い性能が得られている．したがって，GPU メモリの確保方法の改善や，GPU メモリの確保回数をより削減可能なスケジューリング方法の導入により GPU を増やした際の性能が向上すると期待される．データ移動および GPU メモリ確保のコストを無視した場合の図 15 と比較すると，2GPU において 1GPU より約 1TFLOPS の性能向上が確認された一方，4GPU については若干ではあるが 2GPU よりも低い性能が得られた．

次に問題サイズを問題サイズを 65536 × 65536 とした場合に注目する．32768 × 32768 と同様，データ移動および GPU 上のメモリ確保がコストとなっていることがわかる．また，特に 2 つのコストを無視した場合においてはより多くの GPU を用いることで性能が向上していることが確認できる．したがって，32768 × 32768 では問題サイズが十分ではなかったといえる．さらにデータ移動やメモリ確保のコストを削減し十分な問題サイズを用いた場合において，より多くの GPU を用いることでより高い性能を得られるということが示された．

表 5: 手法 2 を用いた複数の GPU による評価のメモリコピー回数

memcpy type	1GPU	2GPU	4GPU
Host to Device	960	1451	2418
Device to Host	465	465	465
Device to Device	0	2248	3372
Total	1425 (x1)	4164 (x2.9)	6255 (x4.4)

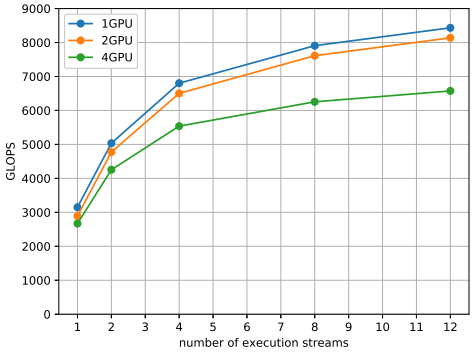

図 10: 問題サイズを 65536x65536 とした場合のコレスキー分解の結果

表 6: 内訳

potrf	白
trsm	緑
syrk	黄色
gemm (行列積) offloading	青
gemm (行列積) task end	灰
memcpy start (H2D)	オレンジ
memcpy start (D2H)	ピンク
memcpy start (D2D)	赤

5. 関連研究

本章では，本研究に関連する研究について記述する．

5.1 OmpSs

OmpSs では CPU のみならず，FPGA や GPU などにオフロードを行うタスク並列モデルにも対応する [7], [8]. OmpSs においてはデバイスを管理するための helper thread を用意し，そのスレッドにオフロードの管理などを集約する．評価においては，Zync SoC FPGA プラットフォームを使用し行列積を用いた計算において FPGA に加え CPU を活用することでより高い性能が得られている．一方，helper thread を生成するためすべてのコアで計算を行わせる場合において helper thread によるコストが隠蔽できず性能に影響を与えることが確認されている．

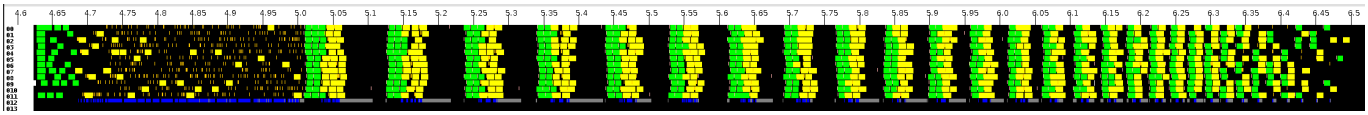

図 11: 1GPU 実行時の可視化

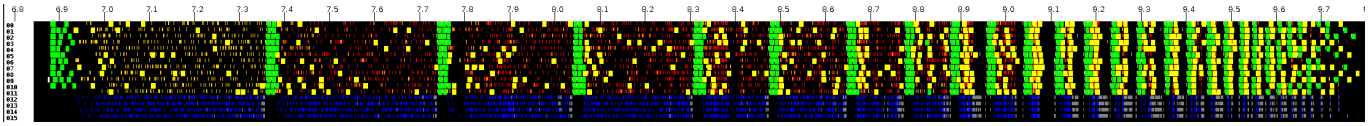

図 12: 4GPU 実行時の可視化

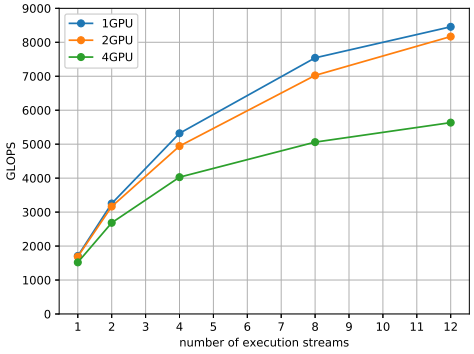

図 13: 問題サイズを 32768x32768 とした場合の，データ移動コストを無視したブロックコレスキー分解の結果

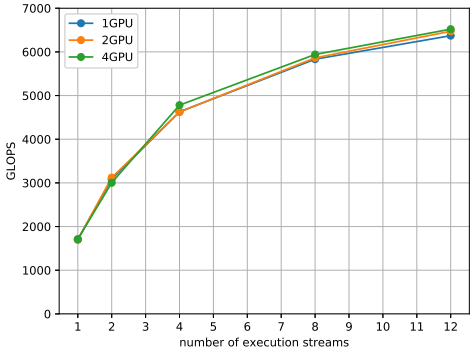

図 14: 問題サイズを 32768x32768 とした場合の, cudaMalloc のコストを無視したブロックコレスキー分解の結果

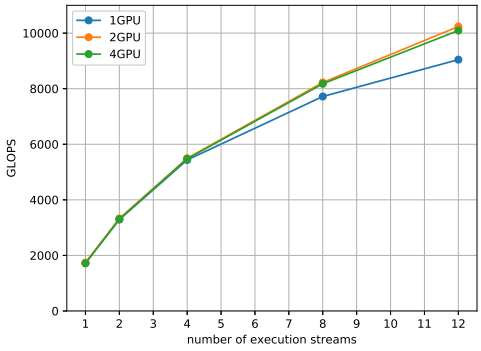

図 15: 問題サイズを 32768x32768 とした場合の，データ移動および cudaMalloc のコストを無視したブロックコレスキー分解の結果

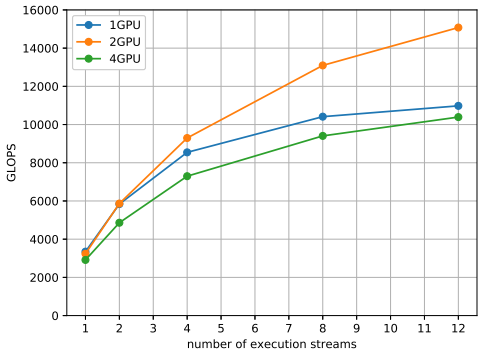

図 16: 問題サイズを 65536x65536 とした場合の，データ移動コストを無視した場合のブロックコレスキー分解の結果

本研究では helper thread を生成するのではなくスレッドそのものを切り替えるという手段を用いている．また OmpSs で用いる Zynq SoC platform は CPU とデバイスがメモリを共有するためデータ移動を行う必要はないと考えられるが，本研究ではメモリ空間が分かれているためメデータ移動を行う必要があり，移動回数 runtime 内で自動最適化する機構を持つ．

5.2 Juggler

Juggler とは米オークリッジ国立研究所による，GPU 向けのタスク並列フレームワークである [9]．OpenMP でタスク並列モデルを記述し，ホスト上でタスク依存グラフを生成し GPU に転送する．その後，GPU 上において依存グラフに基づきタスク並列実行が行われる．評価においては，タスク並列を用いずに GPU 上で実行した場合と比較し最大 31%の性能改善が確認されている．

GPU 内に閉じることにより，ホスト・デバイス間のデータ転送をタスク実行の前後の 2 回に抑えることができる一方，タスクの実行自体はすべて GPU 内で行うことになり CPU など他のリソースを活用することは容易ではない．また，GPU 内でタスク依存グラフに基づき各タスクを実行するという性質上，複数の GPU を用いて実行することや複数のノードを用いて実行することは容易ではないと考え

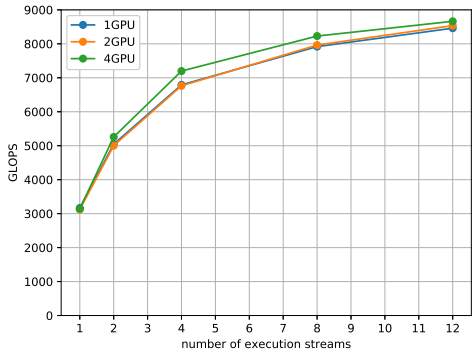

図 17: 問題サイズを 65536x65536 とした場合の, cudaMalloc のコストを無視したブロックコレスキー分解の結果

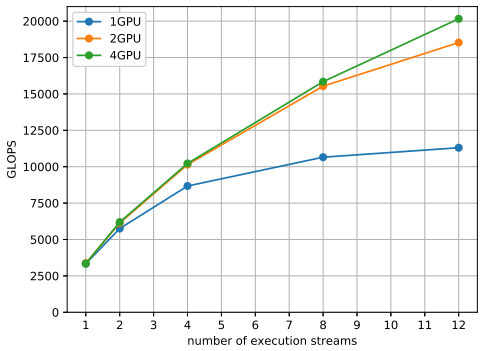

図 18: 問題サイズを 65536x65536 とした場合の，データ移動および cudaMalloc のコストを無視したブロックコピレスキー分解の結果

られる．本研究では，CPU 主導でタスク並列の実行を行うため，ホスト・デバイス間のデータ転送回数が増えるというデメリットがある一方，複数のデバイスを使用することや，通信を含めるといった点については容易に行うことが可能である．

6. 結論

本研究では，演算加速機構を用いてより高いスループットを得るために必要な効率的な実行時システムの提案を行った．効率的に実行するための機構として，「ユーザによる明示的な記述を必要としないホスト・デバイス間通信の最適化機構」を行った．また演算加速機構間のデータ移動の最適化のための予備調査を行った．

ブロックコレスキー分解を用いた評価においては，演算加速機構を 1 つ用いる場合に実行時システムの「ホスト・デバイス間のメモリコピー最適化機構」によりユーザの明示的な記述なしに大幅にメモリコピー回数を削減可能であること，性能向上に寄与することを確認した．問題サイズを 32768×32768，ブロック数を 32×32 とした場合においては，データ移動回数の削減を約 14 倍削減可能であることを確認し，FPGA を用いた場合に約 1.16-1.18 倍，GPU を用いた場合には約 1.6-4.8 倍の性能向上につながった．よって単一の演算加速機構を用いる場合においては提案した実行時システムを用いることで効率的に実行可能であると確認した．複数の GPU を用いる場合においては，ホスト・演算加速機構間データ移動の最適化に加え演算加速機構間の通信手法の違いによる性能への影響について評価を行い，より多くの GPU を用いた場合に性能が低下することを確認した．性能低下の原因についても確認し，データ移動コストの増加や GPU メモリ確保コストが性能低下の原因のうちの 2 つであることを確認した．これらについては設計の改善によるデータ移動コストの隠蔽や「タスクの実行に必要なオブジェクトがどこにあるか」を考慮したスケジューリングの導入により改善可能であると考える．

今後は，データ移動コストのさらなる隠蔽やスケジューリングの改善を行い複数の GPU を用いる場合の性能が改善されるか確認を行う．また，ノード内でのタスク並列実行を拡張し分散メモリ環境における分散タスク並列実行への対応などを進める．さらに，[5] などで述べた指示文をサポートする OpenMP コンパイラの作成を進める．

参考文献

[1] スーパーコンピュータ - 筑波大学 計算科学研究センター https://www.ccs.tsukuba.ac.jp/supercomputer/
[2] V100 - NVIDIA https://www.nvidia.com/en-us/data-center/v100/
[3] Intel Stratix 10 FPGAS - Intel https://www.intel.com/content/www/us/en/products/programmable/fpga/stratix-10.html
[4] Perlmutter - A 2020 Pre-Exascale GPU-accelerated System for NERSC - Architecture and Application Performance Optimization https://waccpd.org/wp-content/uploads/2019/12/NickWright_Keynote_N9_WACCPD_2019.pdf
[5] Watanabe Y., Lee J., Boku T., Sato M. (2018) Trade-Off of Offloading to FPGA in OpenMP Task-Based Programming. In: de Supinski B., Valero-Lara P., Martorell X., Mateo Bellido S., Labarta J. (eds) Evolving OpenMP for Evolving Architectures. IWOMP 2018. Lecture Notes in Computer Science, vol 11128. Springer, Cham
[6] A10PL4 PCIe FPGA Board - Bittware https://www.intel.com/content/www/us/en/products/programmable/fpga/arria-10.html
[7] Filgueras, Antonio, Eduard Gil, Daniel Jimenez-Gonzalez, Carlos Alvarez, Xavier Martorell, Jan Langer, Juanjo Noguera, and Kees Vissers. "Ompss@ zynq all-programmable soc ecosystem." In Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays, pp. 137-146. ACM, 2014.
[8] Jaume Bosch, Antonio Filgueras, Miquel Vidal, Daniel Jimenez-Gonzalez, Carlos Alvarez, and Xavier Martorell. 2017. Exploiting Parallelism on GPUs and FPGAs with OmpSs. In ANDARE '17: 1st Workshop on AutotuniNgSystems, September 9, 2017, Portland, OR, USA. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3152821.3152880
[9] Mehmet E. Belviranli, Seyong Lee, Jeffrey S. Vetter, and Laxmi N. Bhuyan. 2018. Juggler: A Dependence-Aware Task-Based Execution Framework for GPUs. In Proceedings of PPoPP '18: 23nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '18). ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3178487.3178492