Vol.2018-HPC-165 No.25
2018/7/31

OpenMPタスク並列実行における FPGAオフローディングの性能最適化

渡部 裕[1,a)]　李 珍泌[2]　朴 泰祐[3]　佐藤 三久[1,2]

概要：高性能計算の分野において, Field Programmable Gate Array (FPGA) が近年の技術進歩により性能が向上したため、アクセラレータとして注目を集めている. OpenMP 4.0 から導入された, dependency-aware なタスク並列モデルタスク並列モデルでは, タスク間の依存性に基づき生成された task graph に従い実行可能なタスクが並列に実行される. FPGA を用いてタスクベースの並列プログラムを高速化する手法の一つとして, 大量かつ並列に実行可能であり, かつ実行時間が支配的なタスクを FPGA に非同期にオフロードする方法が考えられる. 本研究では,　FPGA にオフロードされた計算について、FPGA リソースを有効に活用するためのトレードオフについて述べる。FPGA を利用するためにループは、OpenCL に変換するが、FPGA のプログラムは直接ハードウェアとなるため, 限られた FPGA のハードウェアを有効に利用するためには OpenCL カーネルの実装方法における「カーネルインスタンスのサイズ」と「カーネルインスタンス数」のトレードオフが必要である. ブロックコレスキー分解を例とし, 支配的なタスクである GEMM をオフロード対象とした評価では, 高性能なカーネルを 1 つ使用する場合と比較し, より小規模なカーネルを複数実装しオフロードした場合に高い性能を得られた. 要因の一部として, 周波数の違いやスループット性能の違いが考えられる.

1. はじめに

近年, 高性能計算の分野で FPGA がアクセラレータとして大きな注目を集めている. 近年のハードウェア技術の進歩により性能が向上しており, いくつかのアプリケーションでは FPGA を活用することで高い演算性能や消費電力性能を得られることが分かっている. 例えば, 最新の Intel 社製 FPGA である “Stratix 10” では, 10 TFlops の理論単精度浮動小数点演算性能を備えておりアクセラレータとして十分な性能を備えている.

一方, Xeon Phi などのメニーコアプロセッサにおける近年のトレンドとして, 並列アプリケーションの開発者のなかで OpenMP[1] を用いたタスク並列プログラミングが注目を集めている. 広く使用されている, OpenMP のループ並列などの work sharing 構造と比較し, タスク並列プログラムを用いることでより高い並列処理性能を得られることもあるためである. タスク並列プログラムでは, 生成されたそれぞれのタスクはタスク間のデータ依存性に基づきスケジューリングが行われたうえで実行される.

OmpSs[2] はタスクプログラミングにおける先駆者として知られており, その機能は OpenMP にも取り込まれている. OmpSs はタスクプログラミングを用いたアクセラレータへのオフローディングに対応している. オフローディング対象として Xeon Phi, GPU, Xilinx FPGA などに対応している. FPGA にタスクをオフロードする場合, OmpSs Runtime は helper スレッドを生成しオフロードタスクの管理を行っている.

本論文では, OpenMP タスク並列を用いたプログラムの一部を非同期に FPGA オフローディングを行う場合において, 限られた FPGA リソースを有効に活用するためオフロードされたタスクを処理するカーネル実装における「カーネルインスタンスあたりの性能」と「カーネルインスタンス数」のトレードオフについて示す. 前提として, OpenMP タスク並列モデルを用いたプログラムを実装し, 特定のタスクを FPGA にオフロードするものとする. FPGA におけるプログラムはハードウェアに変換され, また計算の実行の間に書き換えることは現実的ではないため, 限られた FPGA のリソースをどのように有効に活用するかという最適化問題が発生する. また, FPGA にオフロードするタスクを注意して選択する必要がある. 後述するブロックコレスキー分解を用いた例では, GEMM タスクが最もドミナントなタスクであることからオフロード対象とする. ここで, FPGA へオ

[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

フロードされた GEMM の処理方法については, 次の 2 つが考えられる.

- 高性能なカーネルを一つ実装し, オフロードされた計算を逐次的に処理する
- より小規模なカーネルを複数実装し, オフロードされた計算を並列に処理する

このトレードオフについて, Open Computing Language (OpenCL)[3] を用いて FPGA プログラムを実装し評価を行う. OpenCL とは, アクセラレータを搭載するヘテロジニアスシステムにおけるプログラミング環境を統一することを目的としたフレームワークである. 現在, 多くの CPU, GPU, FPGA ベンダーなどが OpenCL フレームワークを用いた開発をサポートしているが, それぞれの最適化手法は異なる. 本研究では, Intel 製 FPGA 向けアプリケーションを開発するための toolchain である Intel FPGA SDK for OpenCL[4] を利用する. この SDK により, 従来の Verilog などのハードウェア記述言語 (HDL) を用いた開発の困難さが緩和されている. なお, OpenCL を用いて HPC において FPGA を活用するための研究は数多く行われている [5], [6].

ブロックコレスキー分解を用いた評価では, GEMM をオフロード対象とし, 高性能なカーネルを一つ実装した場合と比較し, 小規模なカーネルを複数実装し並列に処理した場合により高い性能を得られた. 本論文における貢献は以下のとおりである.

- OpenMP タスク並列を用いた FPGA への非同期オフローディング手法について示す
- FPGA 上のプログラムを実装する際における, カーネルインスタンスあたりの性能とインスタンス数におけるトレードオフについて示す
- トレードオフについて, FPGA への非同期オフローディングを実装し評価を行う

2. 動機と目的

第 1 章で説明した通り, 本論文では OpenMP タスクモデルを使用し, 一部のタスクをオフロードすることを前提とし, FPGA リソースを有効に活用するための実装方法におけるトレードオフに注目している. コレスキー分解のブロッキングアルゴリズムであるブロックコレスキー分解を用いて評価を行う. コレスキー分解は, ハミルトニアン行列 A を下三角行列 L およびその共益転置である $L^{\dagger}$ に分解を行う計算である. これはタスク並列モデルで記述される典型的なアプリケーションであり, 以下の 4 つの計算により構成される.

- 対称行列の階数 n の更新 (SYRK)
- コレスキー分解 (POTRF)
- 三角行列を係数行列とする行列方程式の求解 (TRSM)
- 行列積の計算 (GEMM)

図 1 は, 5×5 ブロックコレスキー分解のタスクフローを示している. 各ブロックがタスクを表しており, また黒線で in もしくは inout の依存性を示している. ブロックサイズが 5×5 の場合, 20 個のみのタスクが GEMM であることが分かる. ただしブロックサイズが大きくなるにつれ GEMM タスクの数は増加し, 32×32 の場合は全 5984 個のタスクのうち 4960 個のタスクが GEMM となり, 計算におけるドミナントなタスクとなる. そこで今回はブロックサイズを十分に大きくしたうえで GEMM をオフロード対象とする.

OpenMP タスク並列を用いて一部タスクを FPGA にオフロードするプログラムの概要をプログラム 1 に示す. このプログラムでは GEMM タスクを FPGA にオフロードし, 他のタスクはホストで計算を行っている. *write_data_to_fpga* 関数はホストから FPGA へのデータ転送を行う関数, *read_data_from_fpga* 関数は FPGA からホストにデータ転送を行うプログラムである. また, *enquque_request_to_fpga* 関数はホストから FPGA にオフロードの依頼を行う関数である. このプログラムでは, ハミルトニアン行列がすでに FPGA DDR メモリに転送されていることを想定している. 図 1 に示されている通り, GEMM が必要とするデータは TRSM タスクによってのみ更新されることから, GEMM のオフロードを行う直前ではなく TRSM タスク終了時に FPGA へのデータ転送を行うことにより, FPGA ホスト間のデータ転送を削減できる.

GEMM タスクに注目すると, *enqueue_request_to_fpga* 関数や *read_data_from_fpga* の実行後 taskyield により中断されている. ホストでのタスクが中断されている間も, FPGA ではバックグラウンドでオフロードされた計算の実行やデータ転送が行われる. この手法により FPGA へのオフロード処理の非同期化および CPU/FPGA での計算のオーバーラップを実現する. 中断されたタスクが再開された際, オフロードした計算が終了しているか, もしくはデータ転送が終了したか確認を行う. それらが終了していない場合, 再度中断し他のタスクを実行する.

一般的に, 高性能計算においに FPGA を活用する場合には FPGA リソースを最大限使用できるようにカーネルを 1 つ用意し, ホストからオフロードされた処理を逐次的に実行することが多いと考えられる. しかし, FPGA のカーネルの実行は非同期に行えることから, 複数のカーネルインスタンスを並列かつ非同期に実行することが論理的に可能である. そのため, OpenMP タスク並列モデルで大量のタスクをオフロードする場合において, FPGA プログラムの実装に関し「高性能なカーネルを 1 つ実装しオフロードされた所為を逐次的に処理する場合」と「より小規模なカーネルを複数実装しオフロードされた処理を非同期かつ並列に場合」におけるトレードオフを用いた最適化を行うことができる可能性がある.

本論文では, 3.1 節で説明する通り, 実装において OpenCL フレームワークを用いる. また, OpenCL における SIMD 幅

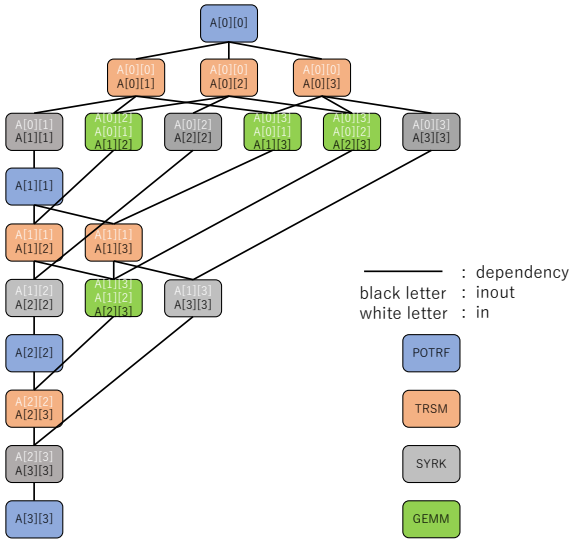

図 1: 5x5 ブロックコレスキー分解のタスクフロー図

をパラメータとしカーネルインスタンスあたりのリソース使用量を調整する. カーネルインスタンスあたりのリソース使用量が小さい場合, 可能な限り複製を行う. カーネル周波数に注目すると, インスタンスあたりの回路がより簡潔である小さなカーネルの周波数はより高くなると考えられる. そのため, 大量のタスクをオフロードした場合, 小規模なカーネルを複数実装した場合により高速に処理できる可能性がある.

このトレードオフを用いた最適化は, ブロックコレスキー分解以外にも同様のタスク並列アプリケーションに適用可能であると考えられる.

プログラム 1: プログラムの概要

```
 1  void cholesky(const int ts, const int nt, float* A[nt
        ][nt])
 2  {
 3  #pragma omp parallel
 4  #pragma omp single
 5    for (int k = 0; k < nt; k++) {
 6  //create POTRF task
 7  #pragma omp task depend(out:A[k][k])
 8      {
 9        omp_potrf(A[k][k], ts, ts);
10      }
11      for (int i = k + 1; i < nt; i++) {
12  //create TRSM task
13  #pragma omp task depend(in:A[k][k]) depend(out:A[k][i
        ])
14        {
15          omp_trsm(A[k][k], A[k][i], ts, ts);
16          write_data_to_fpga(A[k][k], event0);
17          waitForFinish(event0);
18        }
19      }
20      for (int i = k + 1; i < nt; i++) {
21        for (int j = k + 1; j < i; j++) {
22  //create GEMM task
23  #pragma omp task depend(in:A[k][i], A[k][j]) depend(
        out:A[j][i])
24          {
25            enqueue_request_to_fpga(gemmKernel, ..., 
        event1);
26            do {
27  #pragma omp taskyield
28              checkStatus(event1, &ret);
29            } while (ret != done);
30            read_data_from_fpga(A[j][i], event2);
31            do {
32  #pragma omp taskyield
33              checkStatus(event2, &ret);
34            } while (ret != done);
35          }
36        }
37  //create SYRK task
38  #pragma omp task depend(in:A[k][i]) depend(out:A[i][i
        ])
39        {
40          omp_syrk(A[k][i], A[i][i], ts, ts);
41        }
42      }
43    }
44  #pragma omp taskwait
45  }
```

3. 実装

本章では実装について述べる. はじめに実装で用いる OpenCL などに関する説明を行い, 後に実装の詳細について記述する.

3.1 OpenCL

図 2 は OpenCL を用いたアクセラレータプログラミングにおけるオフロードの流れを示している. この例では, OpenCL カーネルが 1 つ, また Command Queue が 1 つ存在する. Command Queue とは, オフロードタスクを管理するための Queue である. ホストプログラムでは, “clEnqueueNDRangeKernel” API を用いてオフロードの依頼を行う. カーネルがオフロードされた計算を実行可能な状態である場合, Queue に入っている 1 つのリクエストが実行される. その後, ホストでは, “clWaitForEvents” API もしくは “clGetEventInfo” API を用いてオフロードした計算の終了確認を行う. “clWaitForEvents” API では, オフロードリクエストが完了するまで待機を行う. “clGetEventInfo” API では, オフロードリクエストの状態を取得することができる.

OpenCL には “single work-item” と “NDRange” の 2 種類のカーネルモデルが存在する. 前者はスレッド並列化を行わずに計算を行うモデルで, 後者は最大 3 次元までのスレッド並列化を行い計算を行うモデルである.

3.2 Intel FPGA SDK for OpenCL

本論文では Intel FPGA SDK for OpenCL を用いて FPGA プログラムおよびホストプログラムにおけるオフロード

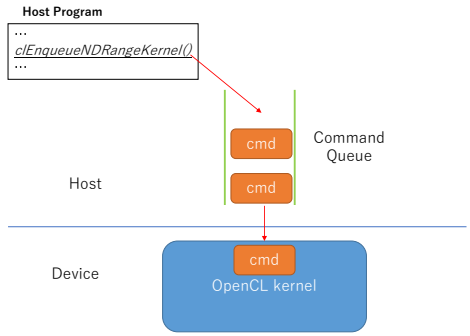

図 2: OpenCL を用いたオフロードの流れ

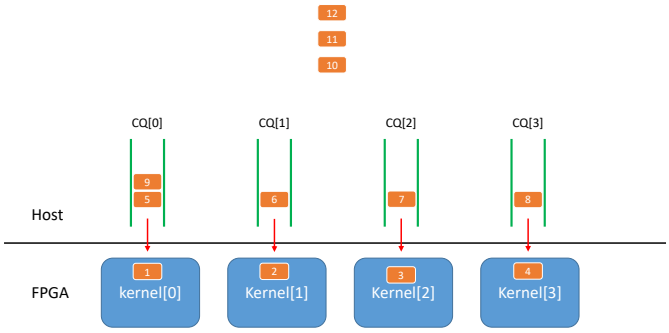

図 3: 複数カーネルインスタンスに対する Queueing

処理の実装を行う. 本ツールチェインは, OpenCL フレームワークを用いて FPGA アプリケーションを実装するためのものであり, OpenCL-to-Verilog の高位合成コンパイラを含んでいる. Board Support Package と呼ばれる, PCIe コントローラやメモリコントローラなどのモジュールを含むライブラリが提供されているため, プログラマはそれらを意識せず使用できる.

本ツールチェインでは, 既存の OpenCL の仕様に対し独自の拡張を加えている. その 1 つの例が, カーネルに対し付与する “num_simd_work_items” 属性である. この属性は NDRange モデルで使用可能であり, SIMD 幅の指定を行う. これらの独自拡張は, Intel FPGA 向け OpenCL プログラムの最適化を行う上で重要な要素である.

3.3 GEMM OpenCL カーネルの実装

オフロードされる GEMM の計算を行うカーネルは, NDRange モデルを用いて実装する. その際, ブロッキングアルゴリズムを使用し, ブロック当たりの大きさを 64×64 とする. DDR メモリへのアクセスを削減するため, 各ブロックを計算時に M20K Block RAM にロードし再利用する. また, “num_simd_work_items” 属性をパラメータとし, カーネルインスタンスあたりのリソース使用量の調整を行う. また, カーネルインスタンスあたりのリソース使用量に応じて複製を行う. 他の最適化として, ツールチェインが提供する 1KB メモインターリブを使用せず, 各データが確保されるメモリバンクの明示を行う.

3.4 タスク並列での非同期オフロードの実装

OpenMP によって生成される GEMM タスクは, FPGA に GEMM 計算のオフロードを行うタスクである. “clEnqueueNDRangeKernel” API を用いてオフロードの依頼を行った後, ホストは taskyield を使用し, 該当する GEMM タスクを中断し別のタスクの実行を行う. ホストの GEMM タスクが中断されている間も FPGA によって各オフロードリクエストの実行が行われる. 中断した GEMM タスクが再度実行された際, “clGetEventInfo” API を活用しオフロードリクエストが完了しているか確認を行う. 完了している場合は FPGA から計算結果の転送を行い終了する. 終了していない場合は再度 taskyield により中断し他のタスクの実行を行う.

複数のカーネルインスタンスが FPGA に実装されている場合, 図 3 のようにオフロードおよび実行を行う. 各カーネルは独立した Command Queue を参照している. ホストの各 GEMM タスクはラウンドロビン形式で Queue を選択する. このように実装することで, それぞれのカーネルは互いに影響を受けず非同期に実行することが可能である.

これらを実装する中で, Intel OpenMP において taskyield directive を使用した場合に期待通りタスクスイッチが行われないことを発見した. これは, OpenMP の仕様において taskyield の挙動が実装依存としていることに起因していると考えられる. そのため, プログラム 1 で記述した疑似プログラムと同じセマンティクスを持つプログラムを, 独自のタスクスケジューリングシステムを使用し実装した. 図 4 に概要を示す.

はじめに, CPU コアに対し OS thread を bind する. その OS スレッドは user-level thread queue を持つ. ランタイムにより user-level thread が生成され, OS thread の user-level thread queue にスケーリングされる. 図のオレンの四角が user-level thread を示す. OS thread は user-level thread queue の常に監視しており, 依存性が解決されている, 実行可能なスレッドがある場合は queue の head から取り出し実行を行う. スレッドが taskyield を要求した場合, OS thread によって中断され queue の tail に戻す. その後, 次の thread が実行される. これらの user-level thread の切り替え処理は OS thread に依存しておらず, user-level API を用いて行われる. 我々はこのシステムを Argobots[7] を用いて実装した. Argobots とは, アルゴンヌ国立研究所にて研究が行われている軽量スレッドおよびタスクフレームワークである. なお, OpenMP でも正しく taskyield が挙動すれば同様のプログラムを実装することが可能である.

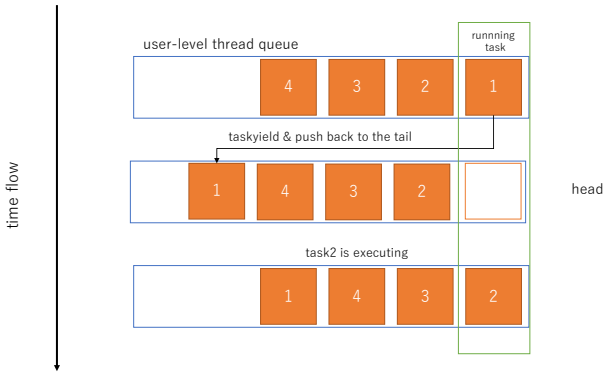

図 4: タスク Queue 実装の概要

表 1: evaluation environment

CPU	Xeon E5-2660 v4 @ 2.00GHz x 2
Host DRAM	DDR4-2400 16GB x 4
GPU	NVIDIA Tesla P100 PCIe x 2
FPGA Board	BittWare A10PL4
FPGA	Intel Arria 10 (10AX115N3F40E2SG)
FPGA DRAM	DDR4-2133 4GB x 2
InfiniBand	Mellanox ConnectX-4 EDR
OS	CentOS 7.3 64bit
FPGA Compiler	Intel FPGA SDK for OpenCL 17.1.2
CPU Compiler	Intel C Compiler 18.0.1
CPU BLAS Library	Intel Math Kernel Library
Thread Library	Argobots 1.0b1

4. 評価

4.1 評価環境

評価では, 筑波大学計算科学研究センターにて運用されている Pre-PACS version 10 (PPX) を使用した. 表 1 に構成の詳細を示す. 本システムは Intel 社の Broadwell CPU を 2 つ, Bittware 社の A10PL4 FPGA[8] ボードを 1 つ搭載している. この FPGA ボードは Intel 社の Arria 10 FPGA[9] を 1 つ, および DDR4 4GB メモリが 2 バンク搭載されている. また, GPU や Infiniband HCA を搭載しているが本実験では未使用である.

4.2 GEMM カーネルのリソース内訳

前述した通り, GEMM カーネルの性能は SIMD 幅およびインスタンス数によって決定される. そこで 3 つの SIMD 幅を使用し 3 種類のカーネルを実装した. それぞれのカーネルは SIMD 幅およびカーネルインスタンス数を除き同じ OpenCL コードを使用している. SIMD 幅について, Type1 は 16, Typ2 は 8, Type3 は 4 と指定した. また, Type1 のカーネルインスタンスは 1 つ, Type2 は 2 つ, Type3 は 4 つである. Type2, Type3 のように複数カーネルインスタンスが実装されている場合, それぞれは並列実行が可能である. Type1 のカーネル周波数は 164 MHz であり, Type2 は 179 MHz, Type3 は 213 MHz である. このように SIMD 幅を小さくする代わりにインスタンス数を増やすことで, より高い周波数が得られている. ただしカーネルの最適化は不十分であり, 最適化によりより高い周波数を得られると考えられる.

表 2: Resource Utilization

	SIMD length	frequency	ALMs	DSP	BRAM
Type1	16	164.71MHz	25%	69%	60%
Type2	8	179.59MHz	27%	69%	69%
Type3	4	213.94MHz	31%	70%	90%

4.3 GEMMk カーネルの評価

はじめに, 基礎評価として各 GEMM カーネルのレイテンシおよびスループットについて, 単精度浮動小数点での評価を行う. この場合においてもタスク並列モデルを使用している. レイテンシは PCIe を経由して行われる呼び出しの時間も含まれる. レイテンシの評価では, カーネルインスタンスが複数実装されている場合も 1 つのみを利用する. スループットでは, すべてのカーネルインスタンスを利用する.

図 5 は各カーネルのレイテンシを示す. ここでは, 4 つの問題サイズを用いて評価を行った. なお, 縦軸は log スケールで表示されている. 評価から, 最も SIMD 幅が大きい Type1 カーネルのレイテンシが最も低いことが分かる. 一方, 問題サイズが大きくなるにつれ, Type3 ではおよそ 4 倍ずつレイテンシが増加しているが, Type1 は 4 倍以上となっている. したがって, Type3 は性能が問題サイズに大きく依存していないが, Type1 では依存しているといえる.

図 6 はカーネルのスループットを示す. 問題サイズが 512×512 の場合, Type1 と Type3 の性能はほぼ同一である. しかし, 問題サイズがより大きい場合は Type3 が最も良い性能を示し, Type1 は最も低い性能を示す. 要因の一つとして, 周波数の違いが考えられる. なお, Type3 の理論性能は 400 GFlops を超えると考えられるが実際に得られた値は 350 GFlops 程度である. 要因との一つとしてホストのタスク管理における最適過不足が考えられ, 改善によりスループット性能も向上すると考えられる.

これらの結果より, 多数のタスクを FPGA にオフロードする場合は高性能なカーネルを一つ実装するのではなくより小規模なカーネルインスタンスを複数実装した場合により高いスループット性能を得られることがわかった.

4.4 FPGA を併用したブロックコレスキー分解の評価

図 7 は GEMM タスクを FPGA にオフロードしたときのブロックコレスキー分解の性能を示している. 単精度浮動小数点を使用している. 縦軸は全体の性能, 横軸は DSP の使用率を示している. Type3 のように複数のカーネルイン

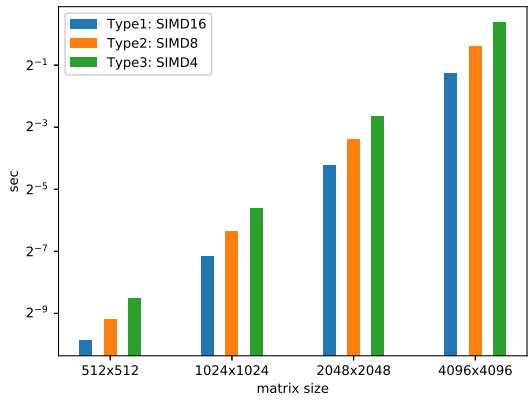

図 5: GEMM カーネルのレイテンシ

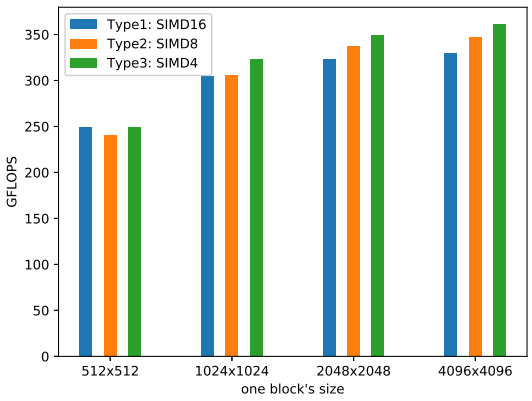

図 6: GEMM カーネルのスループット

スタンスが実装されている場合, 使用するカーネルインスタンス数に応じて複数表示している. ハミルトニアン行列の大きさは 32768 × 32768 とし, 32 × 32 のブロックに分割する.

図で示されている通り, Type3 カーネルを用いた場合に最も高い性能が得られた. Type1 と Type3 の性能を比較すると, スループット性能での性能差は 30 GFlops 程度であるにもかかわらず, Type3 は Type1 に対しおよそ 100 GFlops 高い性能を示している. この結果はカーネル周波数の違いとは別の要因があることを示していると考えられるが, これについては現在調査中である. Type3 の性能に注目すると, カーネルインスタンスの使用数に応じて性能が高まっているが, それほどスケールしていない. これについても, 要因の 1 つとしてタスク管理の最適過不足と考えている. 優先度付き task queue を実装し, オフロードを行うタスクとホストで計算するタスクの重みを変えることで改善できるのではないかと考えている.

全体として, ブロックコレスキー分解の例では FPGA に対し高性能なカーネルを 1 つ実装する場合と比較し, より小規模なカーネルを複数インスタンス実装した場合により高い性能を得られるということを示している.

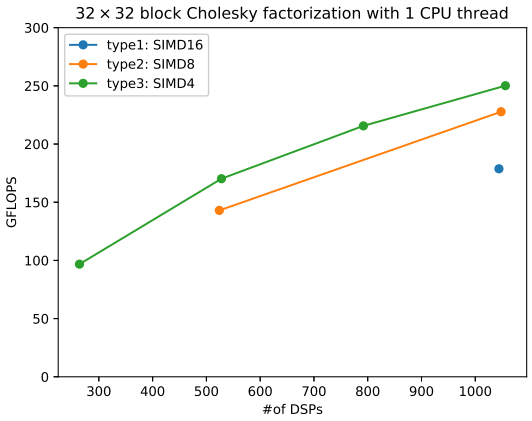

図 7: FPGA を併用したブロックコレスキー分解の結果

5. 関連研究

オークリッジ国立研究所により研究が行われている Open Accelerator Research Compiler (OpenARC)[10] は, GPU や Intel FPGA などへのアクセラレータオフローディングに対応するコンパイラである. FPGA へのオフロードについては, OpenACC プログラミングモデルをベースとし拡張を加えることで対応している [11]. ユーザが記述した OpenACC プログラムを, コンパイラによって OpenCL ホストプログラムおよびデバイスプログラムへ変換を行う. オフロードでは同期モデルを使用している.

バルセロナスーパーコンピューティングセンターにより開発が行われている OmpSs は OpenMP を拡張したフレームワークである. OpenMP に類似した機能に加え, Xeon Phi や GPU, Xilinx Zynq FPGA などのアクセラレータへのオフロードに対応している. Zynq FPGA とは ARM プロセッサを搭載した System on Chip プラットフォームである. FPGA に関しては, ユーザは OmpSs プログラミングモデルに従いプログラムを記述し, コンパイラによって Vivado HLS 向けのプログラムに変換する. ユーザによって Vivado HLS 向けの最適化を行う必要がある. OmpSs runtime によって, FPGA へのオフロードを管理するための helper thread が生成される [12], [13].

OpenARC や OmpSs と異なり, 我々は明示的な非同期実行を使用し FPGA/CPU 計算のオーバーラップを行っている. また, 本論文では FPGA カーネル実装方法におけるトレードオフについて論じている.

6. 結論

FPGA のハードウェアリソース制限により新たな最適化における問題を確認した. これは FPGA におけるプログラムとは直接ハードウェアとなるものであり, CPU や FPGA と異なり実行中に容易に書き換えられるものではないためである. 本論文では, OpenMP タスク並列モデルのなかで

FPGA オフロードを行う場合に関し, FPGA カーネル実装方法における,「高性能なカーネルを 1 つ使用する場合」と「より小規模なカーネルインスタンスを複数用意する場合」におけるトレードオフを用いた最適化について示した.

タスク並列モデルで実装されたブロックコレスキー分解を用いた例では, 多数生成され, かつ並列に実行可能な GEMM タスクをどのようにオフロードし処理を行うかについて実証を行った. その結果, 大規模なカーネルインスタンスを 1 つ実装するのではなくより小規模なカーネルインスタンスを複数実装し GEMM をオフロードした場合により高い性能を得られた. 要因の一つとしては, 後者の場合においてより高いカーネル周波数を得られていることである.

これらの結果については, ブロックコレスキー分解以外のアプリケーションにも適用可能であると考えられる. ブロックコレスキー分解における GEMM タスクのように, ある特定のタスクにおける並列性が十分であり, かつ多数実行されるタスクがある場合などにおいても性能の最適化を行えると考えられる.

7. 謝辞

本研究の一部は，理化学研究所計算科学研究機構と筑波大学計算科学研究センターの共同研究「ポスト京の並列プログラミング環境およびネットワークに関する研究」による.

参考文献

[1] OpenMP http://www.openmp.org/
[2] The OmpSs Programming Model https://pm.bsc.es/ompss
[3] OpenCL Overview. https://www.khronos.org/opencl/
[4] Intel FPGA SDK for OpenCL https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html
[5] Zohouri, Hamid Reza, Naoya Maruyama, Aaron Smith, Motohiko Matsuda, and Satoshi Matsuoka. “Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs.” In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, p. 35. IEEE Press, 2016.
[6] Kobayashi, Ryohei, Yuma Oobata, Norihisa Fujita, Yoshiki Yamaguchi, and Taisuke Boku. “OpenCL-ready High Speed FPGA Network for Reconfigurable High Performance Computing.” In Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region, pp. 192-201. ACM, 2018.
[7] Argobots http://www.argobots.org/
[8] A10PL4 PCIe FPGA Board https://www.bittware.com/fpga/intel/boards/a10pl4/
[9] Arria 10 FPGA https://www.altera.com/products/fpga/arria-series/arria-10/overview.html
[10] Open Accelerator Research Compiler http://ft.ornl.gov/research/openarc
[11] Lee, Seyong, Jungwon Kim, and Jeffrey S. Vetter. “OpenACC to FPGA: A Framework for Directive-Based High-Performance Reconfigurable Computing.” In Parallel and Distributed Processing Symposium, 2016 IEEE International, pp. 544-554. IEEE, 2016.
[12] 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.
[13] Bosch, Jaume, Antonio Filgueras, Miquel Vidal, Daniel Jimenez-Gonzalez, Carlos Alvarez, and Xavier Martorell. “Exploiting Parallelism on GPUs and FPGAs with OmpSs.” In Proceedings of the 1st Workshop on AutotuniNg and aDaptivity AppRoaches for Energy efficient HPC Systems, p. 4. ACM, 2017.
[14] Intel FPGA SDK for OpenCL Programming Guide https://www.altera.com/en_US/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf
[15] Intel FPGA SDK for OpenCL Best Practices Guide https://www.altera.com/en_US/pdfs/literature/hb/opencl-sdk/aocl-best-practices-guide.pdf