Vol.2018-HPC-163 No.25
2018/3/2

FPGA向けストリームプロセッサ生成のための C言語フロントエンドの開発

李 珍泌[1,a)]　上野 知洋[1]　佐藤 三久[1]　佐野 健太郎[1]

概要：半導体の微細化技術が停滞することによってコア数の増加による性能向上が見込めなくなることが予想され、限られた半導体を効率よく利用する専用ハードウェアが注目を集めている。ASIC による専用ハードウェアの実現は長い開発期間と高い開発コストが必要であるため、プログラミング可能な集積回路を持つ FPGA が HPC アプリケーションの専用ハードウェアを実現する手段として大きく注目を集めている。しかし、従来のプログラミングモデルであるハードウェア記述言語や高位合成コンパイラでは生産性と性能の両方を同時に達成することは難しい。本稿ではデータフローモデルによるストリーム計算に特化した FPGA 開発環境 SPGen をハードウェア合成のバックエンドとする C 言語フロントエンドを開発することで限られた用途で高い生産性と性能を両立させる開発環境の実現を目指す。LLVM をベースに開発された C 言語フロントエンド C2SPD は機能を制限した C 言語のサブセットを受け取り、SPGen の独自 DSL である SPD コードに変換する。SPGen が SPD コードから FPGA ハードウェアを生成し、C2SPD がそれを制御する CPU コードを生成することで C 言語によるストリーム計算のハードウェア開発が可能になる。動作検証の実験では C 言語のコードから正常動作する FPGA ハードウェアが生成されることを確認し、最大動作周波数による性能予測を行った。FPGA オンチップメモリのバンド幅 34GB/s に対して合成されたハードウェアは 3.7GB/s の帯域を消費しており、今後 C2SPD によるベクトル化で並列度を向上させることでメモリ性能を改善できると考えられる。

Development of a C Front-end of Stream Processing Generator for FPGAs

Jinpil Lee[1,a)]　Tomohiro Ueno[1]　Mitsuhisa Sato[1]　Kentaro Sano[1]

1. はじめに

高性能計算 (High Performance Computing、HPC) の分野では増え続けるチップの消費電力が問題となっており、動作周波数を抑えて計算コア数を多く持つメニーコアアーキテクチャが広く使われている。しかし、今後半導体の微細化技術が停滞することが予想され、同じ面積のチップに実現できる計算コアの数も限界に近づくと考えられる。このような背景から、限られた半導体を効率よく利用する専用ハードウェアが注目を集めている。特に機械学習の分野では Google 社の Tensor Processing Unit (TPU) をはじめとした多くの専用プロセッサが開発され、実用化されている。専用ハードウェアを Application Specific Integrated Circuit (ASIC) として実現することで半導体をもっとも効率的に利用することができる。しかし、開発期間が長い、開発費用が高い、用途変更が困難であることから用途が限られる。

ASIC による専用ハードウェアと汎用プロセッサの中間的なアプローチとして Field-Programmable Gate Array (FPGA) が挙げられる。FPGA は基本的なコンポーネントである論理ブロックと再構築可能なネットワークで構成される。ネットワークの配線を変えることで ASIC と同等の論理機能を持つ任意のハードウェアを実現することができる。回路面積や電力効率で ASIC に劣るものの、用途に

[1] 理化学研究所 計算科学研究機構
RIKEN Advanced Institute for Computational Science
[a)] jinpil.lee@riken.jp

ⓒ 2018 Information Processing Society of Japan

よってハードウェアを変更可能であることやプログラミングによる専用ハードウェア開発が可能であることから様々なアプリケーションを扱う HPC の分野で大きく注目を集めている。

FPGA の論理機能をプログラミングするためには VHDL や Verilog HDL などのハードウェア記述言語 (Hardware Description Language, HDL) を用いる。これらの言語は C や Pascal などの高水準言語に類似した構文仕様を持つが、実現されるハードウェア機能を強く意識してビットレベルの信号を記述しなければならない。そのため、専門知識を持たない HPC ユーザが使いこなすには敷居が高い。

本稿は FPGA 上でストリーム計算アーキテクチャを実現する開発環境 Stream Processor Generator (SPGen)[1][2] の C 言語フロントエンドの開発について述べる。SPGen は FPGA プログラミングの対象をデータフローモデルによるストリーム計算に限定し、独自の Domain Specific Language (DSL) である Stream Processing Description (SPD) 言語 [3] でハードウェアを記述することで高い生産性と性能を両立させることを目標に開発が進められている。SPD 言語は HPC 分野におけるデータフロー計算に特化した言語であり、FPGA に効率よく実装されることを考慮して設計されている。しかし、記述方法には制約があり、FPGA の制御のために CPU 側でランタイム関数の呼び出しによる冗長な作業が必要であるなど生産性の面で課題が多い。本稿は SPGen の SPD 言語を汎用プロセッサにおけるマシン語と同様に捉えて、C 言語から SPD 言語への変換を行うフロントエンドの開発を行うことで SPGen の生産性を向上させることを目指す。

本稿の構成は次のようである。第 2 章では関連研究を挙げ、本研究との違いを述べる。第 3 章では背景知識としてストリーム計算技術言語 SPD の概要や制約事項について述べる。第 4 章では C や Fortran などの高水準言語から SPD コードを生成するためのプログラミングモデルの提案を行う。第 5 章では高水準言語コードから SPD 言語コードを生成するコード変換手法や処理系の実装について述べる。第 6 章では本稿で開発を行った処理系を用いて C 言語コードから FPGA で動作可能な回路を生成し、動作検証を行う。第 7 章では結論と今後の課題について述べる。

2. 関連研究

近年、高水準言語を用いた高位合成コンパイラや OpenCL[4] のような汎用プログラミングモデルが使えるようになったことで HPC を含めた様々な分野で注目を受けている。Xilinx Vivado HLS[5] や SystemC[6]、NEC の CyberWorkBench[7] などを用いることで C 言語を用いた FPGA のプログラミングが可能である。しかし、これらのプログラミングモデルはシステムのモデリングや DSP などの組み込み機能をメインターゲットにしたものであり、HPC アプリケーションを扱うには適していない。

OpenCL は GPU などのアクセラレータ向けに設計されたプログラミングモデルであり、Intel FPGA SDK for OpenCL (旧 Altera OpenCL)[8] を用いることで Intel 社の FPGA のプログラミングが可能である。しかし、計算部分を OpenCL の独自構文によるカーネル関数として移植し、その制御は冗長な OpenCL ランタイムの呼び出しが必要であるため、プログラミングコストが高い。また、GPU などのアクセラレータを前提とした並列プログラミングモデルであるため、FPGA で生成されるハードウェアを意識したパフォーマンスチューニングが難しいという問題もある。

Oak Ridge National Lab の Seyong Lee らは指示文による並列プログラミングモデル OpenACC から FPGA ハードウェアを生成する処理系 [9] の開発を行っている。C 言語に指示文を挿入するというわずかな変更でハードウェアの生成と制御のためのランタイム呼び出しが行われるため、生産性が高い反面、FPGA ハードウェアの生成を OpenCL によって行うため、パフォーマンスチューニングの難しさは OpenCL と同様である。

Maxeler 社の MaxCompiler[10] はデータフローモデルに特化した FPGA ハードウェアの生成をサポートする。DSL によるプログラミングで適用範囲を限定し、性能のいいハードウェア生成を目指したものである。しかし、Java のサブセットによって行われるデータフロープログラミングは従来の HPC アプリケーションから大幅なコード修正が必要であるため、生産性が高いとは言えないのが現状である。

本稿で開発を行う C フロントエンドは Maxeler 社のアプローチと類似したものである。バックエンドである SPGen で FPGA ハードウェアが生成可能なコードパターン (ストリーム計算) のみを対象にし、C 言語のサブセットによるストリーム計算のプログラミングをサポートする。MaxCompiler ではクラスライブラリとして提供されるストリーム要素間の参照や再利用をループ文の中の配列参照と依存関係の解析で自動生成するなど、従来の C 言語のコードから少ないコード修正で FPGA プログラミングを行うことを目標とする。

3. ストリーム計算記述言語 SPD の概要

本章ではストリーム計算の記述に特化した DSL である SPD の実行モデルとプログラミングモデルについて述べる。SPD で記述される計算モジュールは 1 次元の連続した要素で構成されるデータストリームを扱う。入力ポートからストリームの一つの要素を受け取って計算を行い、出力を生成する。これを入力ストリームの先頭からすべての要素に対して行い、同じ要素数の出力ストリームを生成する。SPGen は SPD コードをデータフローグラフに変換して最適化を行い、HDL 言語で記述された計算モジュール

```
1  Name        core;
2  Main_In     {Mi::x1, x2, x3, x4, sop, eop};
3  Main_Out    {Mo::z1, z2, sop, eop};
4  EQU         equ1, t1 = x1 * x2;
5  EQU         equ2, t2 = x3 + x4;
6  EQU         equ3, z1 = t1 + t2;
7  EQU         equ4, z2 = t1 - t2;
8  DRCT        (Mo::sop, Mo::eop) \\
9            = (Mi::sop, Mi::eop);
```

図 1　SPD コード例

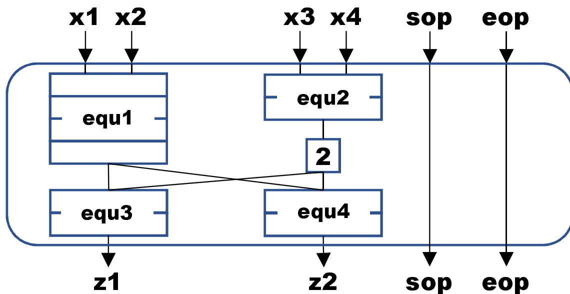

図 2　図 1 から生成される計算モジュールのイメージ

を生成する。

図 1 に SPD のコード例を示す。Name は生成される計算モジュールの名前をあらわす。Main_In は入力ストリームの一つの要素に含まれるデータをあらわす。図 1 では一つの要素に *x1*、*x2*、*x3*、*x4* の 4 つの値が含まれる[*1]。Main_Out は出力ストリームを構成するデータをあらわす。EQU は計算式が与えられ、一つの計算式で一つの計算ノードを構成する[*2]。DRCT で入力と出力を直接接続させることができる。図 1 では制御信号をバイパスするために用いられる。

図 2 に図 1 から SPGen が生成する計算モジュールの概要を示す。EQU で与えられる式が計算ノードとして合成され、入力から出力までのデータの流れがノード間の接続によって実現される。演算のレイテンシを隠し、スループットを向上させるために各ノードがパイプライン化される (図 2 の例では足し算のレイテンシを 2、掛け算のレイテンシの 6 と仮定する)。ノード *equ1* とノード *equ2* のレイテンシが異なるため、2 サイクルの遅延ノードが挿入される。パイプライン化の結果、計算モジュールは毎サイクルで一つの要素を処理する。第 6 章で述べるようにベクトル化を行うことでスループットを向上させることも可能である。

SPD 言語はデータフローを計算式で記述し、FPGA で性能の高いハードウェアを実現することが可能であるものの、記述の制約や生産性に課題がある。EQU ノードの計算式はすべての変数が一つの定義のみを持つ Static Single Assignment (SSA) の形式で記述しなければならない。既存のアプリケーションを SPD 言語に移植するためには大量の変数書き換えが必要になり、これらの変数の導入はヒューマンエラーによるバグを生む可能性がある。また、ストリームの要素間の参照や条件文などが HDL モジュールの関数呼び出しによって実装されるため C 言語などに比べると記述するコード量が増え、直感的ではない。SPGen によって実現されるハードウェアを動作させるためには PCI を経由した CPU メモリと FPGA メモリの間のデータ転送が必要であるが、現状ではベンダー API を用いて実装されるランタイムライブラリ関数による制御が必要であるため、アプリケーション毎に冗長な関数呼び出しが必要である[*3]。

[*1] sop と eop はデータストリームの制御のために用いられる信号である。
[*2] SPGen によって複数のノードをまとめて回路規模を小さくする最適化が行われる。

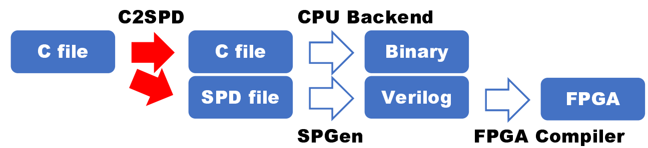

図 3　C 言語から FPGA ハードウェアを生成する処理系の概要

4. SPD 向け処理系の設計

本研究の目標は SPGen を FPGA ハードウェア生成のバックエンドとする C 言語フロントエンドを開発し、C 言語によるストリーム計算ハードウェアのプログラミングを可能にすることである。本章では C 言語から FPGA ハードウェアを生成する処理系の概要とストリーム計算を対象にすることによるプログラミング上の制約について述べる。

4.1 コード変換の流れ

図 3 に C 言語フロントエンド C2SPD と FPGA バックエンド SPGen を組み合わせた処理系の概要を示す。処理系は C 言語のソースコードを受け取り、その一部を取り出して SPD コードを生成する。SPD コードは SPGen によって HDL コードに変換され、ベンダーの論理合成コンパイラによって FPGA 上で動作可能な計算モジュールを実現する。C 言語のコードには FPGA 化された計算モジュールを動作させ、元のコードを CPU で実行した時と同じ結果が得られるようにランタイムライブラリ関数の呼び出しが挿入される。その後、CPU バックエンドによって実行可能なバイナリが生成される。本稿で開発を行う C 言語フロントエンドは入力コードである C 言語コードから SPD コードを生成し、元のコードにランタイムライブラリ関数を挿入するまでの変換を行う。

4.2 対象ハードウェア構成

現在多くの FPGA プラットフォームが PCI インターフェイスで接続可能なカードタイプで提供されている。こ

[*3] OpenCL にも同様の問題が存在する。

```
1  extern float a[1024][1024];
2  extern float b[1024][1024];
3  extern float c[1024][1024];
4  extern float d[1024][1024];
5
6  void calc(void) {
7    for (int i = 0; i < M; i++) {
8      for (int j = 0; j < N; j++) {
9        float temp_a = a[i][j];
10       float temp_b = b[i][j];
11       float sub = (temp_a - temp_b);
12       temp_a = temp_a - sub;
13       temp_b = temp_b - sub;
14       c[i][j] = sub;
15       d[i][j] = (temp_a * temp_a)
16               + (temp_b * temp_b);
17     }
18   }
19 }
```

図 4　C 言語で記述されたストリーム計算の例

のようなプラットフォームはカード上にホストメモリと独立し、FPGA チップから直接アクセス可能なメモリが実装される。FPGA 上に合成されたストリーム計算コアにデータを与えるためには PCI インターフェイスを経由してホスト側から FPGA 側のメモリにデータを転送し、FPGA 上のメモリ間で DMA 転送によるデータストリームの生成を行わなければならない。本稿で開発を行う処理系もカードタイプの FPGA プラットフォームを想定し、PCI を経由するデータの転送や DMA 操作を行う。SPGen は複数の FPGA を想定した開発が可能であるが、本稿では一つの FPGA チップのみを対象にする。

4.3　プログラミングモデルと制約

C2SPD は SPGen の C 言語フロントエンドとして高性能なストリーム計算コアを生成することを目的とする。そのために従来の C 言語にいくつかの制約を設ける。以下に C2SPD が持つ制約の一覧を示す:

- ストリーム計算は配列の参照を行うループ文で記述される
- ループ文の開始・終了条件などは整数の定数で与えられ、コンパイル時に解析できる
- データ配列への参照以外の副作用は存在しない

図 4 に C 言語で記述された C2SPD の入力コードの例を示す。*calc()* の中の 2 重ループ文は各イテレーションで配列 a と b を参照し、配列 c と d に書き込みを行う。そのため、ループ文の計算を配列 a と b で構成される入力ストリームを受け取り、配列 c と d で構成される出力ストリームを生成するストリーム計算として考えることができる。

現在のプログラミングモデルでは C2SPD に与えられたすべてのループ文を SPD に変換する。そのため、図 4 では SPD コードに変換してほしいループ文だけをユーザが関数化 (*calc()*) して個別のファイルとして C2SPD に与えるようにしている。データ配列も元の C 言語ファイルに定義されているため、*extern* キーワードによる宣言が行われている。入出力データの特定やデータストリームの生成を含むすべてのコード変換はコンパイラによって自動的に行われるため、ユーザが対象ループ文を修正する必要はない。

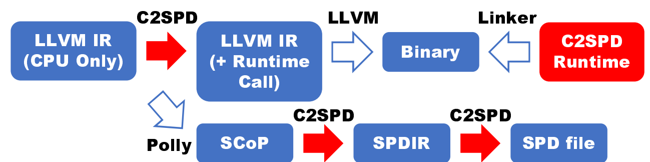

図 5　フロントエンドによる SPD コード生成とホスト側コード変換

以下の制約は現在の SPGen の仕様によるものであり、SPGen の仕様改善によって緩和される見込みである:

- ループ文の実行範囲は配列の宣言範囲と正確に一致する
- 配列のデータ型は単精度浮動小数点のみ
- ループ文の外に宣言されたスカラー変数を利用することはできない (ループ変数は例外)
- ループ文の中で記述される計算は SPD でサポートされる演算のみ (足し算、掛け算など)

5.　フロントエンドのコード変換

本章ではフロントエンドによるコード変換の手法を述べる。C2SPD の開発のためにオープンソースコンパイラである LLVM[11] を用いる。LLVM の C 言語フロントエンド Clang は C 言語コードを LLVM の中間形式である LLVM Intermediate Representation　(IR) に変換する。C2SPD は LLVM IR から SPD コード (FPGA で実行) と FPGA 制御ランタイム関数を含む LLVM IR コード (CPU で実行) を生成する。変換は以下の手順で行われる:

- ループ文の検出と中間形式の生成
- SPD ファイルの生成
- ホスト側のコード変換

図 5 にフロントエンドによる SPD コード生成とホスト側コードの変換の流れを示す。フロントエンドは C 言語で記述されたループ文を検出し、SPD コードと生成する。生成された FPGA 回路を起動させるホスト側コードが元のループ文に置き換わる。

5.1　ループ文の検出と中間形式の生成

C2SPD はループ文の検出や独自の中間形式 SPDIR の生成のために Polly[12] を用いる。Polly は LLVM で利用可能な外部モジュールの一つであり、Z-Polyhedral Model によるループ最適化や SIMD/スレッドレベル並列化を行う。Polly は対象のループ文を解析し、独自の中間形式である Static Static Control Part (SCoP) を生成する。SCoP は LLVM の BasicBlock を一つの statement とし (polyhedral statement)、statement が実行される index の範囲 (domain)

```
1  Name        kernel0;
2  Main_In     {Mi::a, b, sop, eop};
3  Main_Out    {Mo::c, d, sop, eop};
4  EQU         equ0, sub11 = a - b;
5  EQU         equ1, sub12 = a - sub11;
6  EQU         equ2, sub13 = b - sub11;
7  EQU         equ3, c = sub11;
8  EQU         equ4, mul = sub12 * sub12;
9  EQU         equ5, mul18 = sub13 * sub13;
10 EQU         equ6, add = mul + mul18;
11 EQU         equ7, d = add;
12 DRCT        (Mo::sop, Mo::eop) \\
13           = (Mi::sop, Mi::eop);
```

図 6　変換後の計算カーネル (SPD)

やメモリ領域との依存関係 (access relation) を整数の集合で表現する。

C2SPD は SCoP が持つ集合を用いて SPDIR に必要な情報を生成する。SPDIR はループ文の読み込み対象の配列や書き込み対象の配列、データストリームの形状情報、EQU ノードに変換される LLVM 命令のリストを保持する。SCoP の index の domain と access relation を用いてループ文の中 (polyhedral statement で表現される BasicBlock) で参照される配列の領域を計算する。real/write access relation に対して個別に解析を行うことで入力/出力データストリームを生成する。

C2SPD は SCoP 内の BasicBlock が保持するの命令を SPDIR にコピーする。その後、SPDIR 内の命令に対して以下のような解析や変換を行う:

- BasicBlock はループ文の実行部であるため、ループ制御のための命令 (分岐命令や SSA の phi 関数など) を持つ。これらの命令を見つけて削除する。
- 一部のループ制御命令は計算部分と区別が難しい (ループ変数の増加は足し算で実現されるなど) ため、Dead Code Elimination (DCE) 手法によって削除を行う。これによって不要な計算も同時に削除されパフォーマンスが改善する。
- 残った命令がすべて SPD に変換可能であるかどうか検証を行う。SPD でサポートされないデータ型や演算子が存在する場合はエラー終了する。

今後の課題としてデータストリームの変形による性能改善などが考えられるが、そのような最適化は SCoP や LLVM IR の情報を用いて SPDIR 上で実現される予定である。

5.2　SPD コードの生成

図 6 に図 4 のコードから生成された SPD コードを示す。C2SPD は正常な SPDIR から SPD コードを生成する。まず、読み込み、書き込み配列から入力、出力ポートを生成する。*sop*、*eop* はデータ転送などの制御に必要な信号であるため、すべての SPD コードに現れる。

```
1  extern float a[1024][1024];
2  extern float b[1024][1024];
3  extern float c[1024][1024];
4  extern float d[1024][1024];
5
6  void calc(void) {
7    float *__spd_0
8      = __spd_alloc_stream(2097152);
9    float *__spd_1
10     = __spd_alloc_stream(2097152);
11   __spd_pack_contiguous(__spd_0, 0, 2,
12                         (float *)a,
13                         1048576);
14   __spd_pack_contiguous(__spd_0, 1, 2,
15                         (float *)b,
16                         1048576);
17   __spd_run_kernel(__spd_0, 2097152,
18                    __spd_1, 2097152);
19   __spd_unpack_contiguous((float *)c,
20                           1048576,
21                           __spd_1, 0, 2);
22   __spd_unpack_contiguous((float *)d,
23                           1048576,
24                           __spd_1, 1, 2);
25   __spd_free_stream(__spd_0);
26   __spd_free_stream(__spd_1);
27 }
```

図 7　変換後の計算カーネル (C 言語)

EQU ノードの生成は SPDIR が持つ LLVM IR の命令 (Instruction) を SPD 言語の計算式に変換することによって行われる。足し算や掛け算などの演算命令はそのまま計算式に変換される。LLVM IR の演算命令は名前付き仮想レジスタや定数に対してのみ行われる。 また、LLVM IR はレジスタに対する SSA 形式を保証するため、EQU ノードの変数名は代入が行われる仮想レジスタの名前から直接変換するか、適切な prefix をつけることで生成することが可能である。

ロード、ストアなどのメモリ命令はより複雑な処理が必要である。Polyhedtal Statement の実行ドメインとメモリアクセス関数を利用することで、データストリームの依存関係の解析を行うことができる。例えば、ループ文の中で現在のイテレーションより前のイテレーションのデータを参照するメモリ命令が存在する場合、過去に流れたストリームの要素を再利用すると判断し、HDL を用いて実装される SPD のストリーム参照モジュールを生成することが可能である。

5.3　ホスト側のコード変換

図 7 に図 4 のコードから生成されるホスト側 C 言語コードを示す[*4]。ホスト側のコードは PCI インターフェイスを経由したメモリ間のデータ転送や FPGA メモリ上

[*4] 本稿で開発を行った処理系は LLVM の中間言語を用いてコード変換を行うため、実際のホスト側コードは LLVM IR かバイナリファイルに変換される。

で行われる Direct Memory Access (DMA) 操作を行う。このような機能は C2SPD のランタイムライブラリとして実装され、変換されたホスト側コードで呼び出される。*__spd_alloc_stream()*、*__spd_free_stream()* はホスト側のストリームバッファーの確保と解放を行う。対象カーネルは入力ストリームと出力ストリームを持つため、二つのストリームバッファーを生成する。

SPGen によって生成されたストリーム計算コアにデータを与えるためには配列要素の並び替えが必要である。現在の仕様ではデータストリームの中で入力・出力変数が SPD コードで記述された順番通りに連続していなければならない。これを実現するために、Structure Of Array (SOA) から Array Of Structure (AOS) への変換に類似した配列要素の並び替えが行われる。*__spd_pack_contiguous()* は入力配列の各要素をデータストリームの *offset* の倍数の index の場所にコピーする。これによって読み込み対象の配列をメモリバッファーに pack し、入力データストリームを生成する。*__spd_unpack_contiguous()* はデータストリームの *offset* の倍数の index の場所からデータを出力配列にコピーする。これによって出力ストリームのメモリバッファーから書き込み対象の配列に unpack して CPU メモリへの書き込みを完了する。

FPGA 上で実現されたストリーム計算コアはプログラム開始時から終了時まで物理ハードウェアとして動作し続ける。そのため、正しい結果を得るためにはハードウェアにデータを与えて計算が終了するまで同期を行わなければならない。FPGA ボードは PCI インターフェイスで接続された独立したメモリ空間を持つため、ベンダーのプログラミング API を用いたデータ転送を行わなければならない。以上の操作を実現するために、ストリーム計算を実行する *__spd_run_kernel()* は以下の動作を行う[*5]:

- CPU メモリ（入力ストリーム）から FPGA メモリにデータ転送 (PCI 経由)
- FPGA メモリ間の DMA 転送
- FPGA メモリから CPU メモリ（出力ストリーム）にデータ転送 (PCI 経由)

6. 処理系の動作検証

本章では C2SPD と SPGen を組み合わせて C 言語から FPGA ハードウェアを生成する実験を行うことで処理系の動作検証を行う。評価には Terasic 社の DE5a-Net Arria 10 FPGA Development Kit を利用した。表 6 に DE5a-Net のハードウェア仕様を示す。DE5a-Net は PCI Gen3 インターフェイスで接続されるカードタイプの FPGA デバイスである。DDR3 オンボードメモリを持ち、最大メモリバンド幅は 34GB/s である。

[*5] 各データ転送は正常終了を保証するために同期をとらなければならないが、現在の SPGen やドライバーの制約で停止するサイクル数を明示的に指定しなければならない。

表 1 DE5a-Net のハードウェア仕様

項目	仕様
FPGA	Altera Arria 10 GX FPGA
Memory	DDR3-2133 ×2
	17.067 GB/s ×2
Interface	PCIe Gen3 ×8 Edge

図 4 のコードを C2SPD に与えて出力された SPD コード (図 6) を SPDGen でコンパイルして Verilog HDL コードに変換する。Altera の論理合成コンパイラを用いて FPGA 上でハードウェアを生成し、動作が正常に完了することを確認した。

表 6 に生成されたハードウェアのリソース消費量を示す。ALM は論理ブロックの数、Reg はレジスタの本数、M20k はチップ上の 20kB メモリブロックの消費量、DSP は浮動小数点演算に用いられる Digital Signal Processor の個数をあらわす。Total はハードウエア全体のリソース消費量をあらわす。Kernel は SPD コードで記述された計算部分が消費するリソース量を示す。FPGA は評価環境のリソース量を示す。T/F、K/F は Total と Kernel の FPGA リソース量に対する割合をあらわしたものである (単位: %)。Total と Kernel の差分は常時生成される周辺回路によるもので、SPD コードによってリソース消費量が大きく変わることはない。評価に用いた図 6 のコードは非常にシンプルであるため、リソースの消費は 1%未満に抑えられており、FPGA の制御のための周辺回路と合わせても 8%未満である。

現在の SPDGen ランタイムの実装では FPGA 上での DMA 転送の終了時間をサイクル単位で明示的に指定しなければならない。C2SPD のランタイム関数 *__spd_run_kernel()* には DMA 転送が正しく行われるように十分な待ち時間を定数として与えている。そのため、実行時間を測定するために CPU タイマー関数を挿入しても DMA 関数に指定するサイクル数に影響され、正確な性能評価を行うことはできない。本章ではその代わりに生成

表 2 生成されたハードウェアのリソース消費量

項目	Total	Kernel	FPGA	T/F	K/F
ALM	31845	454	427200	7.45	0.11
Reg	68667	433	1708800	4.02	0.03
M20k	425	31	54260	0.78	0.06
DSP	6	6	1518	0.40	0.40

表 3 生成されたハードウェアの動作スペック

項目	値
パイプラインステージ	22 段
最大動作周波数	231.21 MHz
データストリーム幅	8 Bytes (入力/出力)
要求メモリバンド幅	1.85 GB/s ×2

されたハードウェアの最大動作周波数やデータストリームの幅を用いて性能予測を行った。図 4 のコードはメモリ参照に対する演算数が少ないため、性能はメモリバンド幅律速になると予想される。

表 6 に生成されたハードウェアの動作スペックを示す。SPD コードで記述される計算によって計算モジュールが合成され、スループット向上のためのパイプライン化が行われる。その結果、図 6 は 22 段のステージでパイプライン化される。そのうち最も長いレイテンシを持つステージによって最大動作周波数 231.21Mhz という値が決まる。図 6 のコードは入力、出力ストリームともに二つの単精度浮動小数点変数で構成されるのでデータストリームの幅は 8 バイトであり。パイプライン化によってデータストリームのスループットはサイクル当たり 1 になるので、オンチップ DDR3 メモリに対する要求帯域は 3.7GB/s になる。

生成されたハードウェアの要求メモリバンド幅は DE5a-Net のオンチップメモリのバンド幅である 34GB/s の 1/9 程度である。最大動作周波数は SPD の計算式とそれを実現する FPGA の論理ブロックによって決まるため、動作周波数のみによる要求メモリバンド幅の向上には限界がある。図 6 から生成される計算モジュールは 1 サイクルで一つのデータストリーム要素を処理するが、複数の要素を処理するようにベクトル化を行うことでアプリケーションの要求メモリバンド幅を向上させることができる。SPD のプログラミングモデルでは入出力ポートや計算式を処理する要素数と同じ数だけ複製することでベクトル化を行う。C2SPD でベクトル化を実現するためにはループ内で独立したイテレーションを見つけて unroll を行うことでループ内実行文の複製を行うことが考えられる。このようなコード変更は CPU コードのベクトル化と類似しているため、今後の課題として既存のベクトル化手法の応用により実現する。

7. 結論と今後の課題

本稿では FPGA 上で高性能ストリーム計算を実現する高位合成コンパイラ SPGen の C 言語フロントエンドの開発を行った。機能をストリーム計算に限定することで合成される回路の規模や性能が予測可能であり、HPC 分野で広く使われる高水準言語のフロントエンドを導入することで FPGA プログラミングの生産性が向上する。また、ベクトル化のような従来の CPU 処理系の技術を応用することで合成されたハードウェアの性能改善ができると考えられる。コード変換は特定のプログラミング言語に依存しない LLVM の中間形式で行われるため、各種フロントエンドで LLVM IR に変換される他の言語にも本稿で提案されたコード変換を適用することが可能である。

今後の課題として以下のようなものが挙げられる:

- SPD コード内のベクトル化によるメモリバンド幅の活用
- 配列参照と依存関係の解析によるストリーム要素の再利用と先読みの実装
- 既存の SPD アプリケーションの C 言語移植による性能評価
- 倍精度浮動小数点データ型の導入
- ループ変数モジュールの導入による実行ドメインの制限緩和
- SPD 化の範囲指定や FPGA メモリ上のポインター交換などを記述するプログラミングモデルの設計

参考文献

[1] Sano, K., Suzuki, H., Ito, R., Ueno, T. and Yamamoto, S.: Stream Processor Generator for HPC to Embedded Applications on FPGA-based System Platform, *CoRR*, Vol. abs/1408.5386 (online), available from ⟨http://arxiv.org/abs/1408.5386⟩ (2014).
[2] 航平長洲，健太郎佐野：FPGA によるデータフロー計算機におけるハードウェア資源割当て最適化，技術報告 25，東北大学大学院情報科学研究科, 東北大学大学院情報科学研究科 (2018).
[3] 健太郎佐野，涼　伊藤，啓介菅原：階層的モジュール設計を可能とするストリーム計算コア高位合成コンパイラ (リコンフィギャラブルシステム)，電子情報通信学会技術研究報告 = IEICE technical report : 信学技報，Vol. 115, No. 109, pp. 159–164（オンライン），入手先 ⟨https://ci.nii.ac.jp/naid/40020523875/⟩ (2015).
[4] OpenCL Overview: https://www.khronos.org/opencl.
[5] Xilinx Vivado HLS: https://japan.xilinx.com/products/design-tools/vivado/integration/esl-design.html.
[6] SystemC: http://www.accellera.org/downloads/standards/systemc.
[7] CyberWorkBench: http://jpn.nec.com/cyberworkbench/index.html.
[8] Intel FPGA SDK for OpenCL: https://www.altera.co.jp/products/design-software/embedded-software-developers/opencl/overview.html.
[9] Lee, S., Kim, J. and Vetter, J. S.: OpenACC to FPGA: A Framework for Directive-Based High-Performance Reconfigurable Computing, *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 544–554 (2016).
[10] Milutinovic, V., Salom, J., Trifunovic, N. and Giorgi, R.: *Guide to DataFlow Supercomputing: Basic Concepts, Case Studies, and a Detailed Example*, Springer Publishing Company, Incorporated (2015).
[11] LLVM Compiler Infrastructure: https://llvm.org.
[12] GROSSER, T., GROESSLINGER, A. and LENGAUER, C.: POLLY - PERFORMING POLYHEDRAL OPTIMIZATIONS ON A LOW-LEVEL INTERMEDIATE REPRESENTATION, *Parallel Processing Letters*, Vol. 22, No. 04, p. 1250010 (online), DOI: 10.1142/S0129626412500107 (2012).

ⓒ 2018 Information Processing Society of Japan