一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
VLD2021-69,CPSY2021-38,RECONF2021-77(2022-01)

HPC向けRIKEN CGRAのためのコンパイル環境整備と予備評価

小島　拓也†　Carlos Cesar Cortes Torres††　Boma Adhi††　Yiyu　Tan††　佐野健太郎††

† 東京大学　情報理工学系研究科　113–8654 東京都文京区本郷 7 丁目 3–1
†† 理化学研究所 計算科学研究センター　650–0047 兵庫県神戸市中央区港島南町 7-1-26
E-mail: †tkojima@hal.ipc.i.u-tokyo.ac.jp, ††{carlos.cortestorres,boma.adhi,tan.yiyu,kentaro.sano}@riken.jp

あらまし　Coarse-Grained Reconfigurable Architecture (CGRA) は 2 次元アレイ状に並べられた演算器を核に持ち、計算データフローをマッピングすることで効率的に処理を行う非ノイマン型コンピュータの一種である。そのスループットの高さから CGRA は高性能コンピューティングの計算基盤として注目を集めている。本研究では CGRA 向け OpenMP フロントエンドを実装し、データフローの抽出とそのマッピングを行うコンパイラ基盤の開発を行う。生成されたデータフローグラフの構造について予備評価を実施し、Tree Height Reduction を適用することで、同等のデータフローグラフを維持したままマッピングの省面積化やレイテンシ差を半分以下に削減することできることを明らかにした。

キーワード　CGRA, OpenMP, コンパイラ

1. はじめに

汎用プロセッサはこれまでに、十数段にも及ぶパイプライン化、アウトオブオーダ機構、分岐予測技術などを用いて命令レベル並列性を高め、性能向上を図ってきた。デナード則の終焉により汎用プロセッサの設計はメニーコア化へ方針を変えたが、ノイマン型アーキテクチャが抱える性能上のボトルネックやアムダールの法則に基づく高速化限界などの問題に直面している。したがって、ムーアの法則が鈍化している現在、汎用プロセッサに代わり並列性をより効率的に扱うことができる計算機が求められている [1]。特に、データフロー型計算機はさまざまなアプリケーションドメインにおいて有望視されている非ノイマン型アーキテクチャである [2].

Coarse-grained reconfigurable architecture(CGRA) は小規模な演算器で構成される Processing Element(PE) を 2 次元アレイ状に多数持つデータフロー型計算機の一種である。PE 間でデータを送受信できるように、PE は相互に接続されたネットワークを形成する。一般に、CGRA で実行される計算処理はデータフローグラフで表現される。グラフにおけるノードは演算を表し、各 PE へ割り付けられる。演算間のデータ依存はグラフのエッジとして表され、対応する PE 同士は配線ネットワークを通して接続される。ゆえに、CGRA は中間データをメモリへ退避する必要がなく、対象の処理に特化したパイプラインを構築することで効率的に計算を行う。このような特徴から CGRA は数百 MOPS/mW の高いエネルギー効率を達成し [3]、主にエッジデバイスや組み込み機器で利用されてきた [4]. しかし、近年は前述した従来のアーキテクチャが抱える問題から HPC ワークロード向けのアクセラレータとしての利用も期待されている [5]〜[7]。

FPGA もデータフロー型計算の原理に基づいた領域特化のハードウェアを実現するプラットフォームとして広く利用されている。しかし、任意の論理関数を実現するために性能やエネルギーの観点で大きなオーバーヘッドを持つ。さらに、ビット単位の再構成粒度は EDA ツールにおける処理を複雑化させ、数日にも及ぶ長時間のコンパイルを必要とする場合がある。CGRA の粗い再構成粒度はコンパイル処理の複雑さを緩和できるため、FPGA 上のオーバーレイアーキテクチャとしても利用される [8], [9].

CGRA において性能のスケーラビリティを得るにはコンパイラが十分な並列性を抽出できる必要がある。加えて、CGRA コンパイラは多様なアーキテクチャに対応できる必要がある。なぜなら、CGRA には演算器の種類や数、相互接続ネットワークのトポロジー、条件分岐の取り扱い、メモリシステムなど設計上のパラメータが多数存在するためである。ハードウェア設計時には対象アプリケーションドメイン毎にこれらの設計空間を入念に探索する必要がある。さらに、既存のソフトウェア資源を最小限の修正で利用できることが求められる。本研究ではこれらの課題を解決する CGRA 向けコンパイラの実現に目指し、OpenMP コンパイラを実装する。本稿では HPC 向け CGRA として提案されている RIKEN CGRA[10] をケーススタディに、実装したコンパイラが生成したデータフローグラフを CGRA へマッピングした評価結果を報告する。

2. CGRA とコンパイル技術

CGRA は再構成の方式の点で 2 種類のアーキテクチャに大別できる。一方はサイクル単位で再構成を行い、VLIW プロセッサで採用されているソフトウェアパイプライン技術を前提とした実行方式である。ADRES[11], FloRA[12] などがこれに分類される。もう一方は、データフローを空間的に PE アレイへ割り付け、マッピングを静的に維持し続ける方式である。この方式では、サイクル単位のような細かな単位での再構成は行わず、メモリからの入力データストリームを PE アレイへ流し込み計算を行う。PACT XPP[13] や Piperench[14] など古くからこの方式の CGRA が存在し、近年でも CMA[15]、Plasticine[5]、SNAFU[3] などのアーキテクチャが提案されている。前者に比べ、空間的にデータフローをマップするため必要となる PE ア

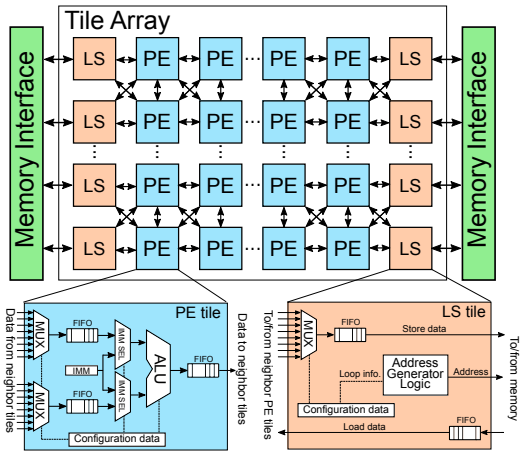


図 1: RIKEN CGRA の構成

レイの数が大きくなってしまうが、再構成のためのオーバーヘッドが小さいためエネルギー効率に優れる。

2.1 RIKEN CGRA

RIKEN CGRA[10] は HPC 向け CGRA として提案されている。現在は、パラメタライズされた SystemVerilog の設計テンプレートとなっており、広域的な設計探索を可能にしている。図 1 にその構成を示す。メモリからのデータ入出力を担当する Load/Store タイル (LS) と演算を担当する PE タイルで構成される。PE タイルは整数および浮動少数点数の算術演算や、論理演算などを行う ALU を持ち、コンフィギュレーションデータで実行する演算を指示する。ALU の入力部には FIFO があり、必要なオペランドが到着すると演算が実行され、結果は出力部にある FIFO へ格納される。この計算結果をオペランドとして必要とする PE の入力 FIFO に空きができるまで、計算結果は出力部の FIFO でキューされ続ける。したがって、RIKEN CGRA も前述の分類では後者のカテゴリに分類されるアーキテクチャである。

LS タイルはメモリからのロード命令もしくはメモリへのストア命令のどちらかとしてコンフィギュレーションされる。前者の場合は PE アレイへのデータ供給を担い、後者の場合は計算結果をメモリへ書き戻す処理を行う。いずれの場合も、Address Generetor (AG) と呼ぶメモリアクセスパターンに基づくアドレス生成ロジックを利用して、メモリアクセスのアドレスを決定する。したがって、コンフィギュレーションデータにはメモリアクセスパターンとして、ループ範囲やインクリメント量などの情報が含まれている。

先行研究の多くでは CGRA の性能評価においてメモリシステムの考慮が不十分である場合が多く、システムとして達成可能な計算スループットについては疑問の余地があった。そこで、RIKEN CGRA は DRAM を用いたメモリシステムを考慮したシミュレーションによる評価が行われており、HBM2 や GDDR6 などの高バンド幅な DRAM を用いることで 10-170GOPS 程度の高いスループットを達成することが示されている [10]。

2.2 関連研究: CGRA 向けコンパイラ技術

CGRA におけるコンパイル処理は大きく分けて以下のステップに分けられる:

（1） フロントエンド処理: ソースコードからデータフローグラフを生成

（2） バックエンド処理: データフローグラフを PE アレイへマッピング (配置配線)

これに加えて、ホストプロセッサから制御されるアクセラレータとして利用される場合は、データ転送やコンフィギュレーションなどの制御ルーチン生成などが必要となる。

2.2.1 フロントエンド技術

これまでに提案されている CGRA 向けコンパイラフロントエンドには CGRA で処理させる箇所を C/C++ 言語のソースコードにおいてプラグマなどで指示する方法 [5], [16]〜[18] や Python などベースにしたドメイン固有言語 (DSL) でデータフローをそのまま記述する方法 [2], [19]、OpenCL のような並列プログラミングモデルを用いる方法 [20] がある。プラグマなどのディレクティブによる方法や OpenCL は既存のソフトウェア資源が利用できる可能性が高く、CGRA に向けに修正するコストが小さい。一方で、提案されているほとんどの手法が特定の方式の CGRA を念頭に置いているため、他の CGRA での利用や、アーキテクチャの設計探索には利用できない。反対に、DSL による方法は設計探索フレームワークに組み込まれている場合が多いが、データフローをそのまま記述するような形式であり、既存ソフトウェア資源の再利用性は極めて低い。したがって、設計探索するを行うためにそのアプリケーションドメインのベンチマークセットを用意するコストが非常に高いという問題がある。

2.2.2 バックエンド技術

CGRA の再構成粒度はワード単位であるため、ビット単位で再構成を行う FPGA で必要な論理合成やテクノロジマッピングなどの処理 [21] が不要である。しかしながら、フロントエンドで生成されたデータフローグラフを PE アレイへマッピングする処理は依然として複雑な処理である。先行研究によって入力されたデータフローグラフにおける有効なマッピングが存在するかどうかの判定は NP-完全であることが示されている [22]. そのため、特定の実行モデルに焦点を当てたヒューリスティック [23], [24] や設計探索を支援する汎用的なアルゴリズム [25]〜[27] など様々なマッピング手法や最適化技術が存在する。しかし、標準的に利用されているフロントエンドが存在しないため、これらの研究でベンチマークとして利用されているデータフローグラフがどのように生成されているかについては明らかではないという問題がある。

3. CGRA OpenMP コンパイラ

3.1 設計方針

本研究では、既存のコンパイラフロントエンドが抱える課題に対処するために、図 2 に示すコンパイル環境の整備を目指す。本環境は次のような特徴を持つ。

- CGRA における処理の指示には OpenMP の target ディレクティブを使用
- 対象とする CGRA モデルに基づいた解析とデータフローグラフ生成
- CGRA 制御用ランタイムのテンプレートによるカスタマイズ性

特に、本稿では主に図中の点線で囲った箇所の実装し、その評価を報告する。

OpenMP ではバージョン 4.0 以降で GPU などのアクセラ

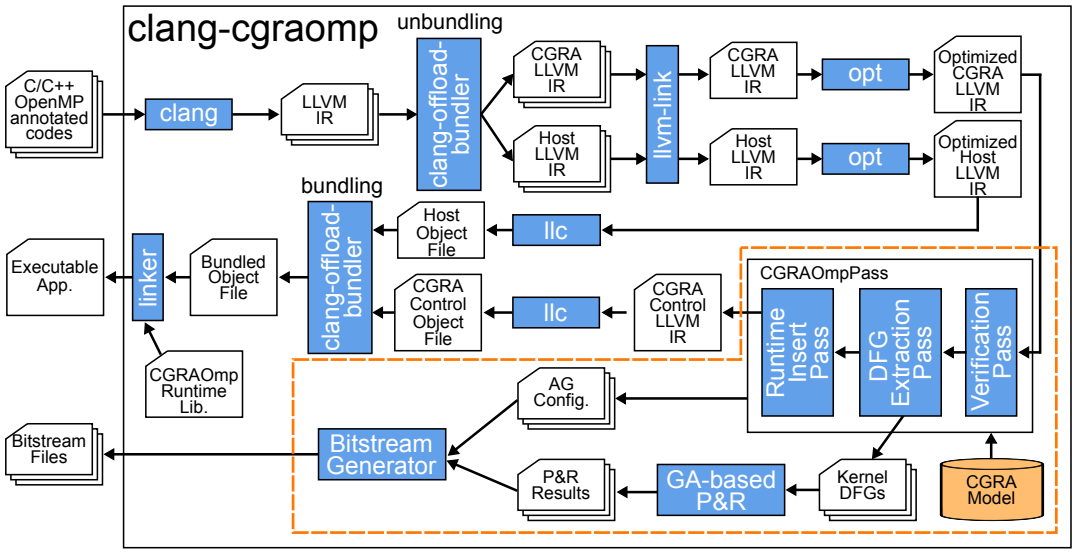


図 2: 開発中のコンパイルフロー．点線部が本研究で新たに実装、評価した部分

レータへ処理をオフロードするための target ディレクティブが追加された。本研究ではこのディレクティブに CGRA を対応させ、ソースコードの再利用性を維持しながらコンパイラが並列性の抽出やデータ転送制御の解析を容易にする。さらに、同様の記述を用いて対象の計算カーネルを GPU で処理させることができるため、これらのデバイスとの比較が可能となる。

本環境を実現するにあたりコンパイラフレームワーク LLVM[28] をベースに開発を行う。LLVM には C/C++ フロントエンドの clang が組み込まれており、ソースコードの字句解析、構文解析、意味解析がすでに完了した中間表現 LLVM IR 形式を扱うことができる。また、LLVM では OpenMP の実装も公開されており、OpenMP で記述されたアプリケーションをホストプロセッサ側と CGRA 側で実行される処理に分離するソフトウェア (clang-offload-bundler) も提供されている．

LLVM では最適化や解析の機能をパスという単位で実装し、パスマネージャーがこれらをパイプライン的に適用する。無用コード除去や共通部分式、定数畳み込みなどの一般的な最適化パスや、別名解析、ループ解析などの解析パスが実装されていおり、一から実装する必要がない。分離された OpenMP カーネルはまず、これらの汎用的な最適化パスを適用し十分な最適化を施す。最適化された IR は本研究で実装するパス CGRAOmpPass によって CGRA 向けのフロントエンド処理を行う。CGRAOmpPass は次のようなパスのシーケンスとして設計する。

（1） **Verification Pass**
- CGRA の実行モデルに基づいてカーネルを解析
- 対象とする CGRA で利用可能な処理かどうかを検証

（2） **DFG Extraction Pass**
- CGRA へオフロード可能な計算カーネルから データフローグラフを生成

（3） **Runtime Insert Pass**
- カーネル内のデータフローグラフとして抽出された 処理部は制御用ラインタイムに置換

抽出されたデータフローグラフはマッピングを行うバックエンドへ入力され、コンフィギュレーションデータを生成する。制御用ランタイムは CGRA との間のデータ転送や、コンフィギュレーションなどの各種制御を行う。そのため、ランタイムが挿入された CGRA 制御コードはホストプロセッサ用のコードとしてリンクされる。ただし、現時点で適切なランタイムルーチンの設計を検討中であるため、Runtime Insert Pass の実装は完了していない。

Code 1: モデルの記述例 (RIKEN CGRA)

```
{
    "category": "decoupled",
    "address_generator": {
        "control": "affine",
        "max_nested_level": 3
    },
    "conditional" : {
        "allowed": false
    },
    "inter-loop-dependency": {
        "allowed": false
    },
    "generic_instructions": [
        "add", "sub", "mul", "udiv",
            ↪ "sdiv", "and", "or",
            ↪ "xor", "fadd", "fsub",
            ↪ "fmul", "fdiv" ],
    "custom_instructions": [ "fexp",
        ↪ "fsin", "fcos", "fpow"],
    "instruction_map": [
        { "inst": "xor", "operand": {
            ↪ "ConstantInt" : -1},
            ↪     "map": "not"},
        { "inst": "xor", "map": "xor"}
    ]
}
```

3.2 CGRA モデルの記述

提案するコンパイル環境では、CGRA モデルに応じて計算カーネルの検証およびデータフローグラフの生成を行う。CGRA モデルは Code 1 を例に示すような JSON 形式で記述される。**category** は CGRA の実行モデルの分類で、本研究ではまず *decoupled* と呼ぶ実行モデルへの対応を行なった。この実

行モデルはメモリアクセスと演算を分離したアーキテクチャモデルに基づくもので、文献 [2] で提唱されている。本研究でケーススタディとする RIKEN CGRA はこの実行モデルに分類される。分離されたメモリアクセスにおいて表現可能なアクセスパターンに関しては **address_generator** で指定される。本研究では RIKEN CGRA の ST タイルが扱うアフィンアクセスに対応した。**conditional** は PE が Predicate 機構などを備え、条件付き処理が可能であるかを示している。対応する CGRA に対してはベクタ化などで利用される if 変換を用いて制御フローがデータフローへ変換される。**inter-loop-dependency** はループ間依存がある場合、依存するイテレーション間でデータ交換を行うハードウェア機構が存在するかを示している。対応する CGRA に対しては、イテレーション間のデータ依存はデータフローグラフにおいては特別なエッジとして表現され、いくつ離れたイテレーションに依存しているかなどの付加的な情報を持つ。

残りの項目は CGRA で利用可能な演算種類に関する記述である。LLVM をベースにしたフロントエンドの多くが LLVM IR における命令の粒度でデータフローグラフを生成する [18], [25]. しかし、これでは LLVM IR に存在しない演算を持つ CGRA に対して有効なデータフローグラフを生成できない。そこで、本モデル記述では **generic_instructions** に LLVM IR の命令中で対応するもののリストを指定し、それ以外の特殊演算 (例えば浮動小数点数の sin 関数など) は **custom_instructions** に列挙する。ソースコード上では所定の属性を付与した同名の関数呼び出しが、データフローグラフの 1 ノードとして抽出される。さらに、LLVM IR 命令のうち所定の条件を満たすものを別の演算のノードとして生成する機能を有する。一般的なプロセッサ ISA においては bitwise not は擬似命令であるが、例として CGRA の ALU に not 演算が実装されている場合には、LLVM IR の xor 命令でかつ、オペランドの一方が定数の “-1” である命令を not 命令のノードとして扱う必要がある。Code 1 の記述例では **instruction_map** にこの変換を行うエントリを記述している。

3.3 データフローグラフ生成と最適化

次に、OpenMP の target ディレクティブを用いた CGRA 計算カーネルの記述について説明する。Code 2 にカーネルサイズ 3x3 の畳み込み演算のコード例を示す。**#pragma omp target parallel for** によって直後の for ループを CGRA での処理対象として指示している。target data ディレクティブである **map** はデータ転送の方向を指示しており、この例では indata 配列のみをホストから転送し、outdata 配列のみを計算終了後に回収すること示している。プログラマが適切にこのディレクティブを設定することで、CGRA とホストプロセッサ間の不要なデータ転送が削減される。この例では畳み込みのフィルタがデータフローグラフにおいて定数ノードとして抽出されることを示すために、ループ中のローカル変数として定義している。また、最内の 2 重ループは各々ループ数は 3 であり、ループ展開されることを想定している。

図 3 は Code 2 を本コンパイル環境に入力し生成されたデータフローグラフである。楕円ノードは演算、矩形ノードは定数、三角ノードはメモリアクセスを表す。ループ展開された LLVM IR の中間表現からデータフローグラフを生成しているため、クリティカルパスが非常に長いグラフとなっている。このような

Code 2: 3x3 畳み込み演算のコード例

```
1  void conv(float * indata, float * outdata)
2  {
3    int irow, icol, orow, ocol, krow, kcol;
4    float tmp;
5    orow = 0;
6    #pragma omp target parallel for map(to:indata
         ) map(from:outdata) private(icol, ocol, orow,
         krow, kcol)
7    for (irow = 0; irow <= (H−K_H); irow += STRIDE
         ) {
8      for (icol = 0, ocol = 0; icol <= (W−K_W); icol
           += STRIDE, ocol++) {
9        const float weight[] = {1.0, 2.0, 3.0, 4.0, 5.0,
             6.0, 7.0, 8.0, 9.0};
10       tmp = ZERO;
11       for (krow = 0; krow < K_H; krow++) {
12         for (kcol = 0; kcol < K_W; kcol++) {
13           tmp += indata[(irow + krow) * W + icol +
                 kcol] * weight[krow * K_W + kcol];
14         }
15       }
16       outdata[orow + O_W + ocol] = tmp;
17     }
18   }
19 }
```

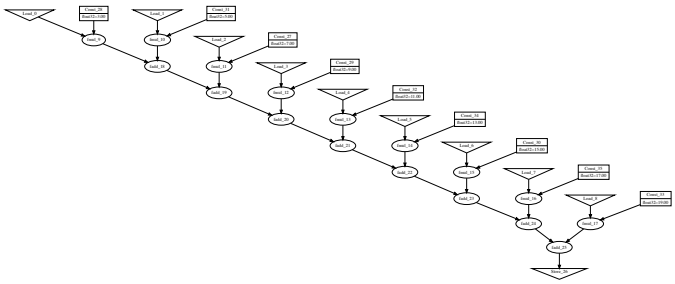


図 3: Code 2 から抽出されたデータフローグラフ

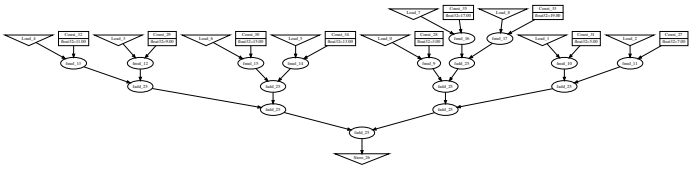


図 4: Tree height reduction 後のデータフローグラフ

グラフは CGRA へマッピングする際に、余計なリソースの消費やレイテンシ、スループットの悪化につながる。そこで、本研究では DFG Extraction Pass にデータフローグラフの構造を変形する最適化を実装する。この最適化ではグラフを木とみなした時に、演算子の交換法則と結合法則に基づいて木の高さを最小化する。これは Tree Height Reduction と呼ばれ、LSI の設計や HDL の高位合成で広く用いられる [29]. さまざまなアルゴリズムが存在するが、本研究ではハフマン符号化のアプローチを用いたアルゴリズム [30] を採用する。この最適化を施した結果を図 4 に示す。ただし、本データフローグラフでは浮動小

表 1: マッピング結果の比較

	面積優先		レイテンシ優先	
グラフ構造最適化	なし	あり	なし	あり
総配線長	43.97	47.89	60.21	46.45
マッピング面積	40	40	56	48
レイテンシ差	8	4	3	1

数点数の加算と乗算が利用されており、一般にこれらは結合法則が成り立たない演算として扱われる。ゆえに、本最適化は浮動小数点数演算に対しては、gcc や clang における fast-math オプションが有効になっている場合のみ適用される。

4. 評　　価

本節では 3. 節で生成したデータフローグラフに対して RIKEN CGRA を対象にマッピングを行い、グラフ構造の最適化がマッピング結果に与える影響を確認する。

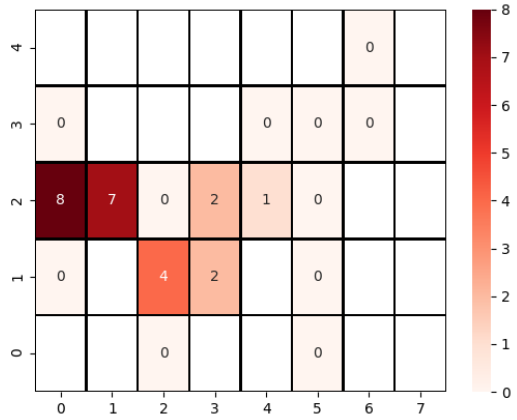
4.1 マッピングアルゴリズム

評価を行うにあたり、RIKEN CGRA 向けにマッピングを行うアルゴリズムを実装する必要がある。そこで、本研究ではオープンソースのマッピング最適化フレームワーク GenMap[27] を RIKEN CGRA へ向けに移植した。GenMap は遺伝的アルゴリズムを用いてマッピングの最適化を行う。最適化の目的関数を複数設定できる多目的最適化が特徴であり、マッピングの最終結果としてパレート最適解を得る。本評価では、最適化項目として 1) 総配線長, 2) マッピング面積, 3) レイテンシ差を設定した。ここで、マッピング面積は利用されている PE を囲む最小矩形の中の PE 数を指し、マッピング高は同様にその高さである。RIKEN CGRA では FIFO によってレイテンシの差を吸収するが、アンバランスな経路が存在すると短い方の経路上の FIFO が埋まりスループットが低下すると考えられる。よって、各 PE で LS タイルから到達する経路のうち最大経路長と最小経路長の差を計算し、PE の中で最大の値を最小化するように目的関数を作成した。

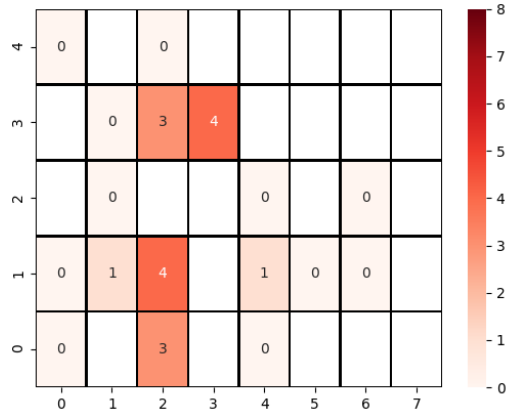
4.2 マッピング結果の比較

図 3 に示したグラフ構造を最適化していないものと、図 4 の最適化を施したデータフローグラフをそれぞれ同じ条件でマッピングし、各目的関数の最適化結果を得た。本評価では得られたパレート最適解の中で、面積が最小のものと、レイテンシ差が最小のものを選択し比較する。その比較結果を表 1 にまとめる。面積を優先した解ではどちらのグラフにおいても面積 40 のマッピングを得たが、グラフ構造を最適化した場合の方が最大のレイテンシ差は半分になっている。一方で、総配線長は若干増加しており、アレイ上での FIFO 利用数の増加を意味する。図 5 は両マッピングにおける各 PE のレイテンシ差分布を示している。各マスは PE に対応しており、レイテンシ値によって色が異なるヒートマップになっている。図 5(a) の左端のようなオペランドが到達する経路に大きな差があると、演算のボトルネックとなり、達成可能な性能に制限があると予想される。

レイテンシを優先した解では、より大きな面積でマッピングをする代わりに各配線を引き伸ばしレイテンシの調整を行う余裕が生まれる。グラフ構造が最適化されていない場合、レイテンシ差は 3 までにしか削減することができないのに対し、最適化を行うと差を 1 まで削減することができている。さらに、グラフ構造の最適化を行なっていると、最小のレイテンシ差を達成するために必要な面積や配線長が小さいことがわかる。以上の結果から本研究で実装したグラフ構造の最適化はマッピングの前処理として有効であると言える。



(a) グラフ構造最適化なし



(b) グラフ構造最適化あり

図 5: 面積優先のマッピングにおけるレイテンシ差の比較

5. おわりに

OpenMP のオフローディング機能を利用したコンパイラ環境の整備を行なった。本研究で整備した環境ではさまざまな方式の CGRA でも利用できるように設計を行い、最初の実装として HPC 向け RIKEN CGRA への対応を目指した。CGRA のコンパイラフロントエンドは計算処理をデータフローグラフとして表現し、抽出する必要がある。しかし、LLVM の中間表現レベルで最適化したデータフローをそのままグラフとして抽出しても、経路長差の点で十分な最適化が行われていないことがわかった。そこで、データフローグラフを CGRA にマッピングする前処理として本研究ではグラフ構造の最適化手法も実装した。グラフ構造の最適化がマッピング結果に及ぼす影響を評価した結果、この最適化を施すことによって PE へ到達する経路長の差を半分以下に削減し、より小さな面積のマッピングを可能に知ることを明らかにした。本評価におけるレイテンシはあくまでマッピングの結果に基づくものであり、実際のシステムで動作させた場合のスループットへの影響を調査するのが今後の課題である。また、本研究で実装した CGRAOmpPass は CGRA モデルに応じて処理を変更可能であるが、現時点では最低限の実装しか行われていない。今後は、多様な実行モデ

ルに対応し、設計探索をできるように機能を拡充していく予定である。

謝　　辞

本研究は、JSPS 科研費 3 次元積層技術を応用した粗粒度再構成可能デバイスの研究 (19J21493)、および国立研究開発法人新エネルギー・産業技術総合開発機構 (NEDO) の委託業務 (JPNP16007) の結果得られたものである。

文　　献

[1] J.L. Hennessy and D.A. Patterson, “A new golden age for computer architecture,” Communications of the ACM, vol.62, no.2, pp.48–60, 2019.

[2] J. Weng, S. Liu, V. Dadu, Z. Wang, P. Shah, and T. Nowatzki, “Dsagen: Synthesizing programmable spatial accelerators,” 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)IEEE, pp.268–281 2020.

[3] G.G.A.O. Atli, K. Mai, B. Lucia, and N. Beckmann, “SNAFU: An Ultra-Low-Power, Energy-Minimal CGRA-Generation Framework and Architecture,” 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)IEEE, pp.1027–1040 2021.

[4] A. Podobas, K. Sano, and S. Matsuoka, “A survey on coarse-grained reconfigurable architectures from a performance perspective,” IEEE Access, vol.8, pp.146719–146743, 2020.

[5] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, “Plasticine: A reconfigurable architecture for parallel patterns,” 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)IEEE, pp.389–402 2017.

[6] Lucas Bragan¥cca DaSilva, R. Ferreira, M. Canesche, M.M. Menezes, M.D. Vieira, J. Penha, P. Jamieson, Jos¥’e Augusto MNacif, “READY: A Fine-Grained Multithreading Overlay Framework for Modern CPU-FPGA Dataflow Applications,” ACM Transactions on Embedded Computing Systems (TECS), vol.18, no.5s, pp.1–20, 2019.

[7] S. Das, N. Sivanandan, K.T. Madhu, S.K. Nandy, and R. Narayan, “RHyMe: REDEFINE Hyper Cell Multicore for Accelerating HPC Kernels,” 2016 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID)IEEE, pp.601–602 2016.

[8] Andr¥’eWerner, F. Fricke, K. Shahin, F. Werner, MichaelH ¥”ubner, “Automatic Toolflow for VCGRA Generation to Enable CGRA Evaluation for Arithmetic Algorithms,” International Symposium on Applied Reconfigurable ComputingSpringer, pp.277–291 2019.

[9] I. Taras and J.H. Anderson, “Impact of FPGA Architecture on Area and Performance of CGRA Overlays,” 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)IEEE, pp.87–95 2019.

[10] A. Podobas, K. Sano, and S. Matsuoka, “A template-based framework for exploring coarse-grained reconfigurable architectures,” 2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)IEEE, pp.1–8 2020.

[11] B. Mei, F.-J. Veredas, and B. Masschelein, “Mapping an H. 264/AVC decoder onto the ADRES reconfigurable architecture,” Field Programmable Logic and Applications, 2005. International Conference onIEEE, pp.622–625 2005.

[12] D. Lee, M. Jo, K. Han, and K. Choi, “FloRA: Coarse-grained reconfigurable architecture with floating-point operation capability,” Field-Programmable Technology, 2009. FPT 2009. International Conference onIEEE, pp.376–379 2009.

[13] M. Petrov, T. Murgan, F. May, M. Vorbach, P. Zipf, and M. Glesner, “The XPP architecture and its co-simulation within the simulink environment,” International Conference on Field Programmable Logic and ApplicationsSpringer, pp.761–770 2004.

[14] H. Schmit, D. Whelihan, A. Tsai, M. Moe, B. Levine, and R.R. Taylor, “PipeRench: A virtualized programmable datapath in 0.18 micron technology,” Custom Integrated Circuits Conference, 2002. Proceedings of the IEEE 2002IEEE, pp.63–66 2002.

[15] T. Kojima, N. Ando, Y. Matshushita, H. Okuhara, N.A.V. Doan, and H. Amano, “Real chip evaluation of a low power CGRA with optimized application mapping,” Proceedings of the 9th International Symposium on Highly-Efficient Accelerators and Reconfigurable TechnologiesACM, p.13 2018.

[16] S. Dave and A. Shrivastava, “CCF: A cgra compilation framework,” 2017. https://github.com/MPSLab-ASU/ccf

[17] A. Ohwada, T. Kojima, and H. Amano, “MENTAI: A Fully Automated CGRA Application Development Environment that Supports Hardware/Software Co-design,” Proc. of 23rd Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI2021), pp.19–24, 2021.

[18] M. Mukherjee, A. Fell, and A. Guha, “DFGenTool: A dataflow graph generation tool for coarse grain reconfigurable architectures,” 2017 30th International Conference on VLSI Design and 2017 16th International Conference on Embedded Systems (VLSID)IEEE, pp.67–72 2017.

[19] C. Tan, C. Xie, A. Li, K.J. Barker, and A. Tumeo, “OpenCGRA: An Open-Source Unified Framework for Modeling, Testing, and Evaluating CGRAs,” 2020 IEEE 38th International Conference on Computer Design (ICCD)IEEE, pp.381–388 2020.

[20] H.-S. Kim, M. Ahn, J.A. Stratton, and W.H. Wen-mei, “Design evaluation of opencl compiler framework for coarse-grained reconfigurable arrays,” 2012 International Conference on Field-Programmable TechnologyIEEE, pp.313–320 2012.

[21] H. Amano, Principles and Structures of FPGAs, Springer, 2018.

[22] M. Hamzeh, A. Shrivastava, and S. Vrudhula, “EPIMap: Using epimorphism to map applications on CGRAs,” Proceedings of the 49th Annual Design Automation ConferenceIEEE, pp.1280–1287 2012.

[23] M. Hamzeh, A. Shrivastava, and S. Vrudhula, “REGIMap: register-aware application mapping on coarse-grained reconfigurable architectures (CGRAs),” Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEEIEEE, pp.1–10 2013.

[24] S. Dave, M. Balasubramanian, and A. Shrivastava, “RAMP: resource-aware mapping for CGRAs,” Proceedings of the 55th Annual Design Automation ConferenceACM, p.127 2018.

[25] M.J. Walker and J.H. Anderson, “Generic connectivity-based CGRA mapping via integer linear programming,” 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)IEEE, pp.65–73 2019.

[26] S. Zheng, K. Zhang, Y. Tian, W. Yin, L. Wang, and X. Zhou, “FastCGRA: A Modeling, Evaluation, and Exploration Platform for Large-Scale Coarse-Grained Reconfigurable Arrays,” 2021 International Conference on Field-Programmable Technology (ICFPT)IEEE, pp.1–10 2021.

[27] T. Kojima, N.A.V. Doan, and H. Amano, “GenMap: A Genetic Algorithmic Approach for Optimizing Spatial Mapping of Coarse-Grained Reconfigurable Architectures,” IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol.28, no.11, pp.2383–2396, 2020.

[28] C. Lattner and V. Adve, “LLVM: A compilation framework for lifelong program analysis & transformation,” Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimizationIEEE Computer Society, p.75 2004.

[29] D.L. Kuck, Structure of Computers and Computations, John Wiley & Sons, Inc., 1978.

[30] K.E. Coons, W. Hunt, B.A. Maher, D. Burger, and K.S. McKinley, Optimal huffman tree-height reduction for instruction-level parallelism, Computer Science Department, University of Texas at Austin, 2008.