

HLS における自動ディレクティブ挿入の一検討

高野 恵輔[†] 井口 寧^{†,††}

[†] 北陸先端科学技術大学院大学 〒923-1211 石川県能美市旭台 1-1

[†] 金沢大学 〒920-1192 金沢市角間町

E-mail: [†]takano@jaist.ac.jp, ^{††}inoguchi@se.kanazawa-u.ac.jp

あらまし 高位合成 (HLS) は C 言語等からハードウェアを生成可能とするが、性能最適化には pragma 等のディレクティブの挿入が不可欠であり、設計者の経験に大きく依存する課題がある。本研究では、C コードを解析しループ構造やメモリアクセス特性に基づいてディレクティブを自動挿入する手法を提案する。複数のベンチマーク回路に適用し、実行時間および資源使用量の観点から評価を行った結果、一定の C 言語記述においては性能向上を達成できることを確認した。

キーワード 高位合成, 自動ディレクティブ挿入, FPGA, 性能最適化

A Study on Automatic Directive Insertion in HLS

Keisuke TAKANO[†] and Yasushi INOUCHI^{†,††}

[†] Japan Advanced Institute of Science and Technology Asahidai 1-1, Nomi-shi, Ishikawa, 923-1211 Japan

[†] Kanazawa University, Kakumachō, Kanazawa-shi, Ishikawa, 920-1192 Japan

E-mail: [†]takano@jaist.ac.jp, ^{††}inoguchi@se.kanazawa-u.ac.jp

Abstract High-level synthesis (HLS) enables the generation of hardware from languages such as C, but performance optimization requires the insertion of directives such as pragmas, which is a challenge that heavily depends on the designer's experience. This study proposes a method for automatically inserting directives based on loop structure and memory access characteristics by analyzing C code. Applied to multiple benchmark circuits and evaluated from the perspective of execution time and resource usage, it was confirmed that performance equivalent to manual design can be achieved for certain C language descriptions.

Key words High-level synthesis, automatic directive insertion, FPGA, performance optimization

1. はじめに

高位合成 (High-Level Synthesis: HLS) は、C/C++ 等の高位記述からハードウェアを生成可能とする設計手法であり、設計生産性の向上に寄与する。特に、近年の FPGA を用いたアクセラレーション環境においては、ソフトウェア開発者によるハードウェア設計を可能にする重要な技術となっている。

一方で、HLS において高性能な回路を実現するためには、ループ展開 (unroll)、パイプライン化 (pipeline)、データフロー最適化 (dataflow) 等のディレクティブ (pragma) の適切な挿入が不可欠である。これらの最適化は設計者の経験や試行錯誤に大きく依存しており、設計コスト増大の要因となっている。

本研究では、C 言語コードを静的解析し、ループ構造およびメモリアクセス特性に基づいて HLS ディレクティブを自動挿入する手法を提案する。提案手法は独自の抽象構文木 (AST) および中間表現 (IR) を用いてコード構造を解析し、対象コードに適した最適化を適用する。

また、本研究では Vitis HLS を対象環境とし、複数のベンチマーク回路に対して性能および資源使用量の観点から評価を行う。

2. 関連研究

従来より、並列化のために OpenMP の pragma 自動挿入に関する研究がなされてきた [1], [2] 同様に HLS における最適化支援に関する研究はいくつか存在する [4], [?] ~ [6]。従来手法は主に以下の 2 つに分類される：

2.1 探索ベース手法

設計空間探索 (DSE) を用いて最適なディレクティブ組み合わせを探索する手法がある。従来はこちらが主流であったが、局所解への収束が課題となっている。AutoDSE [3] は探索ベースの手法であり、非線形最適化を用いて最適なディレクティブ組み合わせを探索する。またこの手法は性能向上に寄与する一方で、探索空間の大きさに起因する計算コストの増大が課題となる。

2.2 ヒューリスティックベース手法

ループ構造や依存関係に基づいてディレクティブを決定する手法が提案されている．これらは実装が比較的容易である一方，最適性の保証が難しい．この手法は単独で用いられることは少なく，探索ベース手法と組み合わせて用いられることが多い．

近年では，LLM を用いたコード解析および最適化支援の研究も進展している [5], [6] ．

2.3 本研究の位置づけ

本研究はヒューリスティックベースに分類されるが，以下の点で特徴を有する：

- 独自 IR による構造抽象化
- メモリアクセスパターンの明示的解析
- LLVM 非依存の軽量設計

3. 提案手法

3.1 システム概要

本システムは，入力された C 言語コードに対して構文解析および中間表現生成を行い，ループ構造およびデータ依存性に基づいてディレクティブを挿入する．従来の HLS 最適化では，pragma の付与位置や種類の選択は設計者の経験に依存していたが，本研究ではこれを静的解析に基づいて自動化する．

処理の流れは以下の通りである：

- プリプロセス（コメント除去）
- 構文解析および AST 生成
- IR 変換（CFG 構築）
- ループ構造解析
- データ依存性解析
- ディレクティブ挿入

特に，本研究では CFG ベースの解析によりループ構造を明示的に抽出し，さらに IR 上での Load/Store 解析を通じてメモリアクセス特性を考慮する点に特徴がある．

3.2 プリプロセス

入力コードに対して，構文解析の前段としてコメント除去処理を行う．C 言語ではコメントが構文解析の妨げとなる場合があるため，本処理によりノイズを排除し，解析の安定性を向上させる．

3.3 構文解析および AST 生成

本システムでは Rust 言語によりパーサを実装し，C コードの構文解析を行う．パーサはパーサコンビネータを用いて構築されており，C11 規格のうち基本的な構文要素（変数宣言，式，制御構文など）に対応している．

解析結果として，独自定義の抽象構文木（AST）を生成する．AST は以下の情報を保持する：

- 文および式の構造
- 制御構文（if 文，for ループ，while ループなど）
- ソースコード位置情報（Span）

特に Span 情報を保持することで，後段の解析結果を元コードへ正確に反映することが可能となる．また，AST はループ構造や制御フローを明示的に表現することで，後段の IR 変換お

よび最適化解析を容易にする．

なお，本パーサは C11 規格のうち以下の構文要素に対応している：

- 基本的な式（算術演算，比較演算，代入）
- 制御構文（if 文，for 文，while 文）
- 配列アクセス

一方で，ポインタ演算や複雑な型修飾子，マクロ展開などの一部構文については対象外としている．これは，本研究の目的が HLS におけるループ最適化に焦点を当てているためであり，対象ドメインを限定することで解析精度と実装の簡潔性を両立している．

3.4 中間表現 (IR) の設計

AST を入力しとし，中間表現（IR）への変換を行う．IR は，基本ブロック（Basic Block）を単位とした制御フローグラフ（CFG）として構成される．各基本ブロックは命令列および終端命令（分岐またはジャンプ）を持ち，これらがエッジによって接続される．

命令は以下の種類に分類される：

- 算術演算命令（Binary）
- メモリアクセス命令（Load / Store）
- 制御命令（Branch / Jump）

特に，本 IR ではメモリアクセスを明示的な命令として扱うことで，配列アクセスの依存関係を解析可能としている．また，一時変数を用いた表現を採用することで，命令間のデータ依存関係を明確化している．

本研究では LLVM 等の既存基盤には依存せず，独自の IR を設計した．これは以下の理由による：

- ツールチェーンのバージョン依存性の回避
- HLS 最適化に特化した情報の明示化
- 軽量かつ拡張容易な解析基盤の実現

IR は基本ブロック単位で構成され，制御フローグラフ（CFG）として表現される．各基本ブロックは命令列および終端命令を持ち，分岐やジャンプにより接続される．

AST から IR への変換過程において，制御構文は明示的な制御フローとして表現される．例えば，if 文は条件分岐とマージブロック，ループ文はヘッダ・本体・退出ブロック，switch 文は分岐列として展開する．この変換により，プログラム構造は CFG として統一的に扱われる．

また生成された CFG に対してループ検出を行い，ループ構造を抽出する．各ループについて 1. ループヘッダ，2. ループ本体，3. 出口ブロック，4. 対応するソースコードの範囲を保持する．これにより，ループ単位での最適化判断が可能となる．

IR においては，Load および Store 命令を明示的に扱うことにより，メモリアクセスの解析が可能となる．これにより変数間のデータ依存関係やメモリアクセス競合，配列アクセスパターンを解析できる．

本研究では，ループの構造とデータ依存関係などの情報を基に，ディレクティブ適用可否の判定を行う．

3.5 最適化解析とディレクティブ挿入

ディレクティブ挿入は，AST に保持されたソースコード位

```

1  #define A 10
2  #define FF(a,b) a * b
3
4  // This is a test file for C11 mini frontend.
5
6  int add(int a, int b) {
7      return a + b;
8  }
9
10 int main(void) {
11     int a = 10;
12     int b = FF(a, 20);
13     float c = 1.0;
14
15     int i, j;
16
17     for(i = 0; i < A; i++) {
18         for(j = 0; j < 3; j++) {
19             a = a + add(i, j);
20             c = c * 1.1;
21         }
22     }
23
24     while (a == 1 || a == 2.2)
25     {
26         b = add(b, 1);
27     }
28
29     return 0;
30 }
31

```

図 1 サンプルのソースコード

Fig. 1 Sample Source Code

表 1 実装環境

Table 1 Implementation Environment

	言語	パーサライブラリ	対象 HLS ツール
環境	Rust	nom	Vitis HLS

置情報 (Span) を用いて実装される。これにより、解析結果を元のコード構造を保ったまま反映することが可能となる。

本研究では以下のヒューリスティックルールに基づいてディレクティブを挿入する：

- ・ 依存関係のないループに対して pipeline を適用
- ・ ループ回数が小さい場合に unroll を適用
- ・ ループ間に依存がない場合に dataflow を適用

3.6 ループ構造の抽出と可視化

ループ構造の抽出は、IR 上での制御フロー解析により行う。具体的には、ループヘッダを特定し、そこからループ本体および出口ブロックを識別する。これにより、ループ単位での最適化判断が可能となる。また、抽出されたループ構造は Graphviz を用いて可視化することができる。これにより、設計者はコードのループ構造を直感的に理解し、最適化の効果を確認することができる (図 1、図 2)。

4. 実装

4.1 実装環境

提案手法は Rust 言語 [8] で実装されており、C コードのパーサには nom ライブラリを使用している。生成されたコードは Vitis HLS でのコンパイルを想定しており、生成されたコードに対して Vitis HLS の最適化オプションを適用して性能評価を行う。

4.2 処理フロー

提案システムの処理フローは以下の通りである：

表 2 評価環境			
Table 2 Evaluation Environment			
	FPGA ボード	HLS ツール	ベンチマークコード
環境	Xilinx Alveo U50	Vitis HLS 2023.2.2 [7]	polybench 4.2.1

- (1) C コードの入力
- (2) AST の生成
- (3) IR への変換
- (4) 最適化解析の実行
- (5) ディレクティブの挿入
- (6) 最終コードの出力と Vitis HLS でのコンパイル

5. 評価

本章では、提案手法を適用した場合のリソース評価および RTL シミュレーションによる評価を行う。対象とする C コードは、BLAS のソースコード (polybench 4.2.1) とし、これらのうち数種に対して提案手法を適用する。C コードは、提案システムが読むことができるように、対象コードに対して必要な前処理 (ヘッダファイルの埋め込み) を行ったものを使用する。

5.1 評価環境

評価環境は表 2 に示す。評価では、提案手法を適用したコードと最適化も適用していないもので性能を比較する。性能評価の指標としては、ステップ数および FPGA のリソース使用量を用いる。性能およびリソース使用量は、Vitis HLS のレポートから取得する。

評価指標として、以下を用いる：

- ・ 実行サイクル数 (Latency)
- ・ FPGA リソース使用量 (LUT, FF, BRAM)

5.2 結果

ベンチマークに使用したソースコードのリソース使用量および動作サイクルを表 3 に示す。結果として、提案手法はベンチマークにおいて、未最適化コードと比較して最大 2800 倍の性能向上を達成した。一方で、今回のベンチマークで利用したコードでは unroll を多用したため、リソース使用量が大幅に増加する傾向が見られた。

元から Cycle 数が低いコードに対しては unroll による性能向上はあまり見られず、逆にリソース使用量の増加が顕著であった。

また今回は、メモリアクセスの最適化が不十分であったため、本来有効であるはずの dataflow や pipeline が適用されなかったことが回路の面積肥大の大きな要因となった。unroll の適用は性能向上に寄与する一方で、リソース使用量の増加を招くため、性能とリソース使用量のトレードオフを考慮しながら、最適化ルールの調整が必要である。

6. まとめ

本研究では、C コード解析に基づく HLS ディレクティブの自動挿入手法を提案した。独自 AST および IR を用いることで、LLVM に依存しない軽量な解析基盤を構築した。評価の結果、提案手法はベンチマークにおいて未最適化コードと比

