東京大学
情報理工学系研究科 創造情報学専攻
修士論文

FPGA 上のグラフ処理のための頂点アクセス並列化を支援する HLS 向けフレームワーク

A Framework for HLS to Support Vertex Access Parallelization for Graph Processing on FPGAs

三冨 秀和
Hidekazu Mitomi

指導教員　千葉 滋 教授

2023 年 1 月

概要

グラフ処理は自律型ロボットの経路探索や計算科学などで用いられており、リアルタイム性や高いエネルギー効率を求めて FPGA 上で処理を行うことがある。効率的なグラフ処理をするための FPGA 回路作成には、ハードウェアの知識が多く要求され一般のプログラマには難しい。これは高位合成ツールを用いることによって緩和されるが、完璧とは言えない。そこで本論文では、間接参照される頂点の BRAM 上の配置がユーザーの書くコードに影響を与えないことを利用して、頂点への並列アクセスを可能にするフレームワークを提案する。本研究で行った提案フレームワークに対する評価により、フレームワークが生成する回路が並列グラフ処理に用いることが可能なこと、そしてフレームワークの入力となるプログラムを高位合成ツールと用いるときより簡潔かつ直感的に記述できることを確認した。

Abstract

Graph processing is widely used in path finding for autonomous robots, computational science, and other applications. Graph processing on FPGAs is being researched for real-time operation and high energy efficiency. In order to create FPGA circuits for efficient graph processing, a lot of hardware knowledge is required, so it is difficult for general programmers. This problem is mitigated by the high-level synthesis tools, but not perfect. In this paper, we propose a framework which supports vertex access parallelization for graph processing on FPGAs, by utilizing that the replacements of vertices' indirect references to BRAMs do not affect the user's code. We confirmed two points through evaluation of the proposed framework. First, the FPGA circuits generated by the framework can be used for parallel graph processing. Second, the program used as input of the framework can be written more simply and intuitively than the high-level synthesis tools.

目次

第 1 章

はじめに

グラフ構造は現実世界のデータをモデル化するのに良く用いられており、それらに関連する洞察を得るためにグラフ処理が用いられる。従来、エッジデバイスでのグラフ処理演算は汎用プロセッサ上で実行されており、消費電力を抑えつつ実行効率を向上することが課題の一つであった。現在、グラフ処理演算に特化したハードウェアに担わせることでこの課題を解決する研究が盛んに行われている。そのようなハードウェアの開発を行う際、回路情報を再構成することができる FPGA を用いることで開発を効率的に行うことが可能となる。

ハードウェア記述言語や高位合成による開発には、ハードウェアに関する知識やメモリ並列化のための工夫が要求されるため、一般のプログラマにはハードルの高いものとなっている。例えば FPGA では演算回路を複数並べることで簡単に演算の並列度を上げることが可能であるが、メモリアクセスがボトルネックとなってしまう場合がある [1]。このような場合では、メモリアクセスの並列度も同時に上げないと処理のスループットを改善することができない。

そこで、本論文では DSL (Domain Specific Language) で記述されたグラフ処理プログラムを入力として、FPGA 上で実行可能にするようなフレームワークを提案する。提案フレームワークは、アルゴリズム中の並列アクセス可能な頂点データを複数の BRAM に配置することによって FPGA 上での並列アクセスを実現する。グラフデータが格納されたメモリに並列にアクセスするために、2 つの BRAM を用いた動作クロック 1 サイクルに付き 2 回頂点データにアクセスすることが可能なメモリコントローラを作成し、フレームワークが提供することとした。

また提案 DSL は、メモリコントローラによるメモリ並列化を意識せず、かつ並列グラフ処理のアルゴリズムを用いなくとも、FPGA 上でのグラフ処理を効率的に行えるようにすることを目標に設計されている。提案 DSL は C 言語のマクロ機能を用いて実装されており、C/C++ 言語の高位合成システム上で利用できる。提案 DSL によって書かれたアルゴリズム中のグラフアクセス部分は、提案メモリコントローラに対するアクセスに自動で変換させる。またグラフ処理の繰り返し演算はユーザーアノテーションを利用して並列処理化されるような設計がされている。

本論文では幅優先探索と Bellman-Ford 法を提案 DSL で実装し、フレームワークによって出力された FPGA 設計を論理シミュレータ上で動作させることによって評価を行った。この

評価によって、シミュレータ上で BRAM に並列アクセスしながらグラフ処理されていることを確認した。また提案 DSL によって 2 つのアルゴリズムを記述した際、高位合成を利用したときよりも簡潔に記述できていることを確認した。グラフ処理の 2 つの典型例において簡潔な記述と並列アクセスの両方を実現できたと言える。

本論文の構成について説明する。第 2 章では、FPGA を用いたグラフ処理の背景と課題について述べる。第 3 章では、提案フレームワークについて特に DSL とメモリコントローラの機能と実装に着目して説明する。第 4 章では、本研究で行った評価内容とその結果について説明する。第 5 章では、本論文のまとめと今後の課題について述べる。

第 2 章

グラフ処理と FPGA を用いたハードウェア設計

グラフ処理はインターネット上の事象の分析や車両・自律ロボットの経路探索など様々な分野で応用されている。特に車両・自律ロボットにおいては、小型デバイスにおいて高速かつリアルタイムに複雑なグラフ処理をしないといけないことがある。これを実現するための FPGA 上のアーキテクチャを用いたグラフ処理について本章で説明する。

2.1 グラフ処理

グラフとは、図 2.1 のように複数の頂点（ノード）とノード間の連結を表す枝（エッジ）で構成されるデータ構造である。グラフ構造をもつデータは World Wide Web (WWW) 、メタゲノミクス、分子結合、分散計算の構造、決定木や自然言語処理のような機械学習で扱うデータなど、さまざまな分野でみられる [2]。グラフ構造から何かの洞察を得るために、グラフ中の要素の関係を調べたり、グラフ要素を探索したりされる。これらを総称してグラフ処理と呼ぶ。例えばウェブページの重要度を決定するアルゴリズム（PageRank[3]）では、WWW 上の各ページのリンク関係をグラフ構造とみなしたデータ上で、ページ間で重要度をやり取りする関係を調べている。あるいは移動ロボットの経路計画問題において、A* アルゴリズムなどのグラフ探索アルゴリズムが用いられることがある [4]。

このようなグラフアルゴリズムの具体例として、まず幅優先探索について紹介する。幅優先探索はグラフ構造に対する探索アルゴリズムの一種であり、グラフの連結部分や重みなしグラフの最短経路を求めるのに用いられる。このアルゴリズムは次のようなフローで行われる。

1. 探索開始ノードをキューに追加する。
2. キューの先頭ノードを取り出す。キューが空ならアルゴリズムを停止する。
3. 取り出したノードの子ノードのうち、未探索のものをキューに追加する。
4. 2. に戻る

次に Bellman-Ford 法 [5] について紹介する。次のような問題を考える。

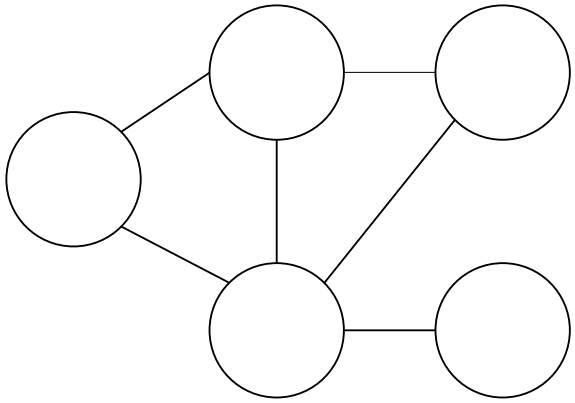


図 2.1: グラフの例

N 個の都市が一方通行の道路で結ばれていて、それぞれの道路を通過するのに必要な時間が与えられている。ある都市から別の都市へ最短の時間で移動するための経路を決定したい。

これは都市をノード、道路をエッジ、通過するのに必要な時間をエッジの重みとした重みあり有向グラフに対して、次のようなアルゴリズムで解くことができる。

1. それぞれのノードのスコア（開始ノードからの距離を表す）を初期化する。開始ノードのスコアを 0 とし、それ以外のノードのスコアを無限大とする。
2. 以下の操作を $N-1$ 回繰り返す。
   （a）辺を一つ取り出す。
   （b）取り出した辺の始点のスコアと、取り出した辺の重みと足す。
   （c）足した値が取り出した辺の終点のスコアより小さかったら、終点のスコアをその値で更新する。
   （d）すべての辺に対して (a) ～ (c) を繰り返す。

Bellman-Ford 法は Dijkstra 法という別の最短経路問題を解くアルゴリズムと比較して、負の閉路を検出できる代わりに計算量が大きいという特徴を持つ。負の閉路とは図 2.2 のようなもので、この場合 $A \to B \to C \to A$ の経路を延々と辿ることでスコアがずっと小さくなり続けてしまう。Bellman-Ford 法では、アルゴリズム終了後にもう一度だけ 2.(a) ～ 2.(d) の操作をした際にスコアに更新がある場合は閉路あり、ない場合は閉路なしという様に検出できる。

2.2　グラフ処理システムのアーキテクチャ

グラフデータには、一般にビッグデータと呼ばれているような大量のノード・エッジのデータを持つものもある。例えば現実のウェブページのリンク関係をグラフにした Hyperlink Graph[6] には、35 億のノードと 1287 億のエッジが含まれている。このような巨大なデータは、CPU 上でシングルスレッドで扱うとメモリ容量が足りなくなったり処理に時間がかかっ

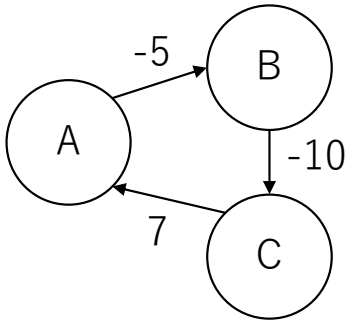
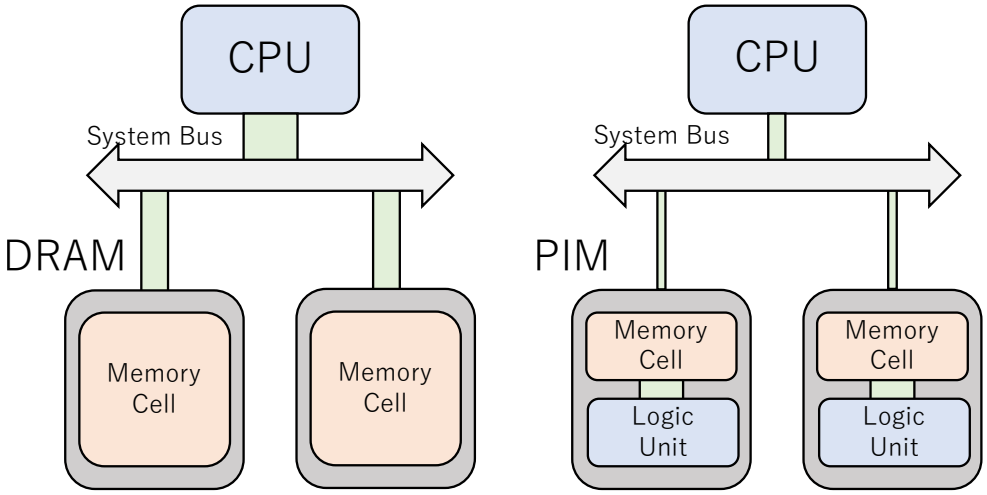


図 2.2: 負の閉路の例



(a) DRAM の場合 (b) PIM の場合

図 2.3: DRAM と PIM の比較

たりするため、グラフ処理システムのアーキテクチャは様々な工夫がされてきた。

まず単一のマシンで処理を実行させる場合について説明する。メモリ使用量を削減するため、グラフ圧縮やグラフをパーティショニングして分割処理することが行われる。例えば PIM-GraphSCC[7] は、PIM というプロセッサを内蔵したメモリを用いている。これは図 2.3 のようにメモリとプロセッサの距離を縮めることで、メモリ帯域を逼迫せず、メモリアクセスのオーバーヘッドを小さくすることができる。グラフを分割したデータを各メモリに配置し、各プロセッサで分散処理している。また実行時間を改善するため、処理対象のグラフやそれに対する課題に応じたヒューリスティックなアルゴリズムの改良がおこなわれる。例えば Lu ら [8] は実ネットワークグラフに対するコミュニティ検出タスクにおいて、最小ラベルヒューリスティックや色分けスキームを用いて実行時間を改善した。

次に分散システムでグラフ処理する研究例について紹介する。1 つ目に Parallel BGL[9] を紹介する。Parallel BGL は Boost's Graph Library を分散グラフ計算向けに拡張したものである。隣接リストの形で表現されたグラフデータを複数のプロセッサに分散配置することができる。また、特定の用途に限定しない汎用プログラミングの考え方の下設計されており、様々

な通信モデルに適応できる。このライブラリは InfiniBand で接続された 128 の計算ノードを有するシステム上で評価された。

2 つ目に Pregel[10] を紹介する。Pregel はクラスタ上で処理することを目的とした分散グラフ処理のためのフレームワークであり、また vertex-centric なグラフ処理モデルを初めて提唱した。vertex-centric モデルでは、各ステップで各頂点に対して実行すべき処理が指定され、処理内ではステップおきに頂点間でメッセージパッシングが行われる。例えば Bellman-Ford 法では各ステップ各頂点ごとに以下の処理が行われる。

1. スコアを含むメッセージを受信し、自身のもつスコアより小さかったらその値で更新する。
2. 近傍頂点に対して、自身の持つスコアと辺の重みを足し合わせたスコアを含むメッセージを送信する。このメッセージは次のステップで受信される。

一般に大規模データの分散処理で用いられる MapReduce[11] 処理系ではすべての頂点データに毎ステップアクセスしなければいけないのに対し、vertex-centric モデルではアクティブな頂点のみが処理対象になる [12]。これは処理するべき頂点にランダム性が大きいグラフ処理においては、頂点データへのメモリアクセスを減らすことができるため適している。さらに上記のような処理は各頂点で独立に実行可能なので、計算ノードに頂点データを分割して配置することによって容易にクラスタで処理することが可能となる [12]。

2.3　専用ハードウェアでのグラフ処理

グラフ処理を専用ハードウェアで行うことは重要である。ハードウェアでグラフ処理を行う利点としてエネルギー効率が良いことが挙げられる。

汎用プロセッサは様々なプログラムを実行できるように設計されている。プログラムには処理の一連を表す命令や命令が用いるデータが含まれており、これらの命令は一般的には様々なプログラムの実行を実現するために細かな演算の粒度になっている。そして動作周波数の 1 サイクルごとに命令が実行されるが、汎用プロセッサの動作を高速化するために周波数は高く設定される。

専用ハードウェアとは特定のアプリケーションを実行するためにつくられた装置のことである。汎用プロセッサのようにアプリケーションが行う処理を命令の形に分解することはしない。その代わり、処理に必要な論理演算を直接回路化する。専用ハードウェアでアプリケーションを実装した場合、処理速度の観点から処理にかかるサイクル数と動作周波数を最適化することが可能である。また、ボトルネックとなる処理を並列に実行するなどの最適化もできる。このため、エネルギー効率という点において専用ハードウェアが汎用プロセッサに勝ることが多い。たとえば、Basic Linear Algebra Subroutines (BLAS) という線形代数ライブラリのうち、行列ベクトル積のエネルギー効率を CPU、GPU、FPGA において比較した研究 [13] では、FPGA が CPU、GPU と比較して同等の性能をより高いエネルギー効率で達成した。

専用ハードウェアでのグラフ処理での研究例として、グラフ処理のメモリアクセスの特色に

注目してメモリ転送量を減らすことで大幅なエネルギー効率上昇に成功した研究 [14] がある。GPU を用いたグラフ処理の研究も数多く行われている [15]。

また、自律型ロボットの経路計画のような組み込み分野 [16] でグラフ処理が用いられることがある。従来は小型汎用プロセッサ上で処理を行ったり、処理をクラウドコンピューティングにオフロードしたりされていた [16]。高速化・リアルタイム化を目的として、性能向上や通信オーバーヘッドの削減のためにアプリケーションに特化したハードウェア上で計算する研究がされている。

2.4 FPGA とハードウェア開発

Field-Programmable Gate Array (FPGA) とは集積回路の一種であり、論理回路を自由に構成・変更できるため、ハードウェアの開発に便利である。FPGA には論理回路のパーツとなる論理ブロックと入出力ピン、それらのパーツを接続するための配線が内蔵されている。開発者は各論理ブロックの動作を決定し、配線を配置することでパーツを組み合わせて論理回路を構成できる。また論理ブロック動作の設定や配線の配置は何度もしなおすことができ、そのため論理回路を変更できる。本節ではより具体的に FPGA について説明する。

2.4.1 FPGA の構成要素

SRAM ベースの FPGA をもとに、FPGA の構成要素について説明する。FPGA の内部には以下のような要素が内蔵されている [17]。

論理ブロック

動作を後から決定することができる論理コンポーネント。一般的な FPGA の論理ブロックははルックアップテーブル（LUT）、マルチプレクサ（MUX）、フリップフロップ（FF）の 3 つから成る。図 2.4 に 2 入力の LUT で AND を表すように構成された論理ブロックを示す。LUT は入力値に対して出力値を割り当てるテーブルである。テーブルとして SRAM が内蔵されており、この情報を書き換えることで動作を決めることができる。図 2.4 の例では、入力値 A1 と A2 が共に 1 のとき出力値 B は 1、それ以外の時は 0 となるように SRAM に情報が書き込まれている。MUX は外部からの信号を受け入れるため、FF は信号の出力をクロックと同期するために存在する。

ブロック RAM（BRAM）

ある程度のサイズ・数を有する SRAM と、SRAM に付帯的な機能を与える周辺回路。以下、Xilinx 社の UltraScale アーキテクチャに搭載されている BRAM をもとに、BRAM の特徴について述べる [18]。各 BRAM は 36kB の領域をもち、独立した 2 つの 16kB の BRAM として動かすことも可能である。各 BRAM は 2 つの独立した読み書きポートを有している。なお、2 つのポートでアドレス競合が発生した場合、両方書き込みだった場合を除いて正常に動作する。1 ポートのみを持つ BRAM の周辺回路を図 2.5 に示す。READ の場合、Read Enable（図では R EN）信号が立てられ、アドレ

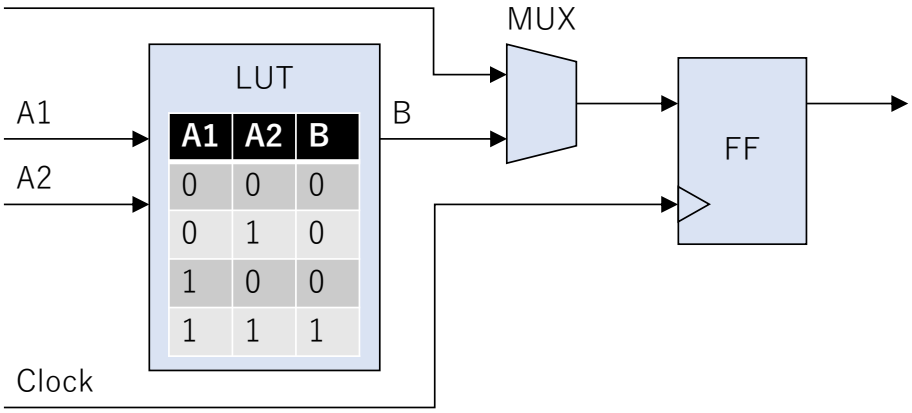


図 2.4: 論理ブロックの論理図

スが指定される。制御回路によって指定されたアドレスにあるデータが SRAM から出力される。後段のラッチは一時的にデータを保存するため、レジスタはクロックに同期してデータを出力するためにある。出力（図では DOUT）の手前にあるセレクタによって、クロックに同期してデータを出力するか否かを選択することができる。WRITE の場合、Write Enable（図では W EN）信号が立てられ、アドレスと書き込むデータ（図では DIN）が指定される。制御回路によって指定されたアドレスに指定されたデータが SRAM に書き込まれる。

入出力ブロック

FPGA の I/O ピンと配線を接続するためのブロック。ピンが入出力のどちらか、プルダウン・プルアップ・オープンドレインの設定をするための制御回路や、PCI のような一般的な通信規格をサポートするための回路が内蔵されている。

配線ブロック

論理ブロックや入出力ブロック、BRAM などの要素を相互接続するためのブロック。

コンフィギュレーションメモリ

論理ブロックの設定や各要素の接続関係を記憶するためのメモリ。

その他の要素

商用 FPGA には上記の要素の他に、加算乗算器や組みこみプロセッサ、デバッグ用回路、より高度な機能を有するメモリなどの機能が内蔵されている。

デバイスによって各要素がどの程度 FPGA に内蔵されているかは異なる。例えば Spartan-7 XC7S50 では表 2.1 のような規模になっている [19]。

2.4.2　FPGA の設計方法

FPGA を用いて論理回路を設計する際の方法について 2 つ紹介する。

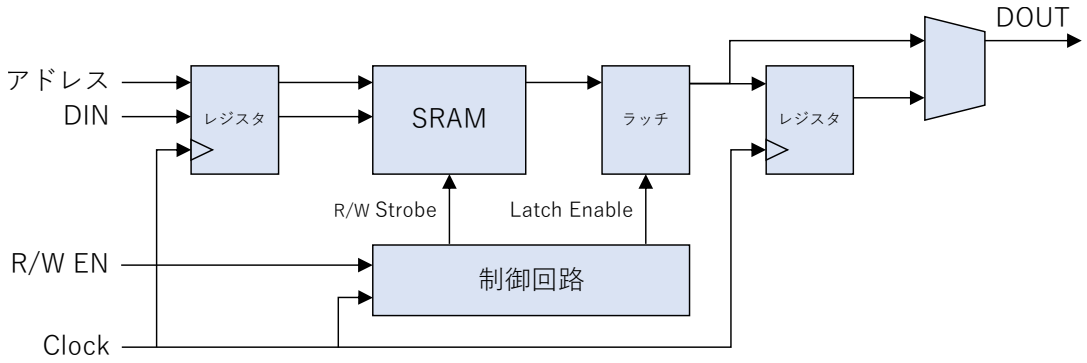


図 2.5: 1 ポート BRAM の論理図

要素	数
論理ブロック	52160 個
BRAM	36kB * 75 個
I/O ピン	250 個

表 2.1: Spartan-7 XC7S50 の各要素の数

ハードウェア記述言語による設計

まず一般的なのが Verilog や VHDL などといったハードウェア記述言語（HDL）を用いた設計である。1980 年代に HDL が登場する以前は、論理回路を論理ゲートの組み合わせとして設計していた。しかし真理値表や論理式、有限状態マシン（FSM）を論理ゲートレベルに変換するのに人手が用いられた。この作業は煩雑で間違いを犯しやすい。そのためより高い抽象度（レジスタ転送レベル、RTL）で論理機能のみを記述し、論理ゲートレベルに変換するのはコンピュータに任せるために開発されたのが、HDL と論理合成ツールである [20]。

Verilog を拡張した SystemVerilog で 4bit 加算器を構成する例をもとに、論理機能の記述方法を説明する。図 2.6 は SystemVerilog による 4bit 加算器の設計の 1 例である。回路を記述する最小単位はモジュールであり、`module` ～ `endmodule` 内で定義する。`FullAdder` は全加算器であり、引数内に入出力信号が定義されている。具体的には入力 `A`、入力 `B`、桁上げ入力 `CarryIn`、出力 `Sum`、桁上げ出力 `CarryOut` の 5 つである。`assign` 文により信号を接続することができる。ここでは出力信号に対して、入力信号をビット演算した値を接続している。モジュール `Adder4bit` は全加算器を 4 つ組み合わせてできた 4bit 加算器である。11 ～ 14 行目で下位モジュールとして先ほど定義した `FullAdder` を 4 回インスタンス化している。このように下位モジュールのインスタンスに信号を割り当てることでより高度な記述が可能となっている。なお、SystemVerilog は図 2.7 のように + 演算子で複数の bit 幅をもつ加算を表現することができる。

HDL コードは論理合成ツールによって、実際の論理ブロックと配線を記述するネットリストに変換される。また、論理合成においてはハードウェアを削減するための最適化もなされ

```
1  module FullAdder(input logic A, B, CarryIn,
2                   output logic Sum, CarryOut);
3      assign Sum = (A ^ B) ^ CarryIn;
4      assign CarryOut = (A & B) | ((A ^ B) & CarryIn);
5  endmodule
6
7  module Adder4bit(input logic [3:0] A, B,
8                   output logic [3:0] Sum,
9                   output logic CarryOut);
10     logic [2:0] carry;
11     FullAdder fa0(A[0], B[0], 1'b0, Sum[0], carry[0]);
12     FullAdder fa1(A[1], B[1], carry[0], Sum[1], carry[1]);
13     FullAdder fa2(A[2], B[2], carry[1], Sum[2], carry[2]);
14     FullAdder fa3(A[3], B[3], carry[2], Sum[3], CarryOut);
15 endmodule
```

図 2.6: SystemVerilog による 4bit 加算器の設計例 1

```
1  module Adder4bit(input logic [3:0] A, B,
2                   output logic [3:0] Sum,
3                   output logic CarryOut);
4      assign {CarryOut, Sum} = A + B;
5  endmodule
```

図 2.7: SystemVerilog による 4bit 加算器の設計例 2

る。例えば図 2.6 の例では桁上げ伝搬加算器を示すネットリストが出力されるが、図 2.7 の例では他の加算器の実装方式（桁上げ先見加算器やプリフィックス加算器）を含めた中から、速度の要求を満たす中で回路規模が最小のものが選択される [20]。

また、論理シミュレーションも HDL による記述が可能である。HDL により設計されたモジュールが正しく動くことをツール上で確認するために用いられる。論理シミュレーションを行うためにはテストベンチを HDL で記述すればよい。例えば 4bit 加算器のテストベンチを図 2.8 に示す。`initial` 文によってシミュレーション開始から実行するべき処理が示される。この場合では、`A` と `B` にある入力を与え 10 単位時間待つ、という処理を繰り返している。論理シミュレーションツールは配線を流れる信号や論理ブロックの状態をシミュレーションし、出力信号がどのようになっているか示す waveform を出力する。waveform は図 4.4 のような見た目をしている。

```
module testbench_Adder4bit();
    logic [3:0] A, B, Sum;
    logic CarryOut;
    Adder4bit adder(A, B, Sum, CarryOut);

    initial begin
        A = 4'b0000; B = 4'b0000; #10;
        A = 4'b0001; B = 4'b0000; #10;
        A = 4'b0010; B = 4'b0000; #10;
        ...
    end
endmodule
```

図 2.8: SystemVerilog による 4bit 加算器のテストベンチ

高位合成による設計

高位合成とは、C++ のような高級言語で記述された回路の機能を HDL に変換する技術である [21]。高級言語という HDL よりも抽象度が高い設計が可能なため、特に複雑なアルゴリズムを FPGA 上で実行したい場合に有用である。他のメリットについても考える。HDL で順序回路を設計する際にクロック単位の最適化を行うには、レジスタ間の信号伝達タイミングや動作周波数を意識しなければならない。また、論理シミュレーションツールによる動作検証は動作が非常に遅く、テストベンチの記述は信号を逐一する必要があるため直感的でない。このような問題に対し高位合成を用いた設計では、クロック単位の最適化は高位合成ツールが担うし、高級言語での記述を CPU 向けの処理系で実行することで動作検証することも可能である。

オープンソースとなっている高位合成ツールの一つである LegUp[22] について紹介する。LegUp は C 言語によって記述されたプログラムを入力とし、プログラムの一部分は FPGA 上で動作するよう設計されたプロセッサ上で実行され、残りの部分はハードウェアとして演算がなされるような回路を出力する。この変換は以下のような順番でなされる。

1:LLVM IR **への変換・最適化**

LLVM IR はさまざまな種類のプログラミング言語のコンパイル基盤となっている LLVM において、実行環境に依存しない中間言語として用いられている。まず C 言語のプログラムを LLVM IR にコンパイルし、その後デッドコードの除去やエイリアス解析などの最適化が行われる。特にエイリアス解析によって、命令が独立であるかどうかのチェックが行われ、したがって並列実行できる部分が見つけ出される。また、プロセッサ上で実行するべき部分と、ハードウェアで演算されるべき部分に分けられる。

2: **デバイスの特性評価とアロケーション**

利用する FPGA デバイスによって、使用可能な回路面積やメモリのサイズ・ビット幅が異なる。これらの特性に合わせて、回路の伝搬遅延、必要な論理素子やレジスタ、乗算器ブロックの数を決定する。

3: **スケジューリング**

スケジューリングでは、プログラム中の各処理をサイクルに割り当てる。まずプログラム中の条件分岐を表す有限状態機械（FSM）を構築する。次に FSM の各ノードに対応する処理ブロックの中で、演算間のデータ依存性と演算の遅延を調べる。その後、各演算ができるだけ早いタイミングでスケジュールされるように割り当てられる。

4: **バインディング**

バインディングによって各演算の演算子が特定のハードウェアユニットに割り当てられ、プログラム中の変数がレジスタに割り当てられる。回路面積削減のために複数の演算子を 1 つのハードウェアユニットにまとめたり、複数の変数を 1 つのレジスタにまとめたりといった最適化がなされる。この時どの演算子・変数を示しているのか区別するためにマルチプレクサが必要となる。マルチプレクサの実装は比較的コストが高く遅延も増えるので、回路面積が削減できるようバランスが取れ、かつ遅延ができるだけ少なくなるような最適化が行われる。

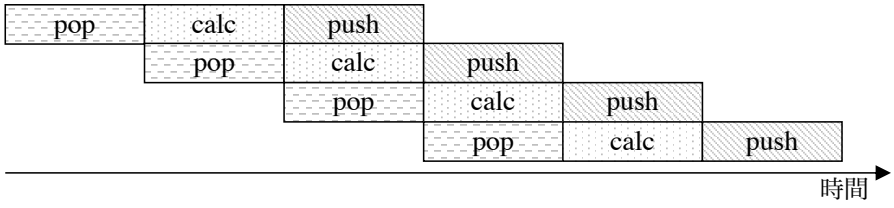
なお、LLVM IR を用いない一般の高位合成ツールにおいても、通常アロケーション、スケジューリング、バインディングの 3 つの処理が行われている [23]。

商用の高位合成ツールとして、Xilinx 社の Vitis HLS[24] を紹介する。Vitis HLS では C++ 言語により記述されたアプリケーションを入力として、ハードウェア記述言語に変換される。このとき Vitis HLS が用意した API の他に OpenCL の API も利用することができる。このツールでも中間表現として LLVM IR を用いている。なお LLVM IR の最適化パスは公開されているが、そこからハードウェア記述言語に変換される部分は公開されていない。

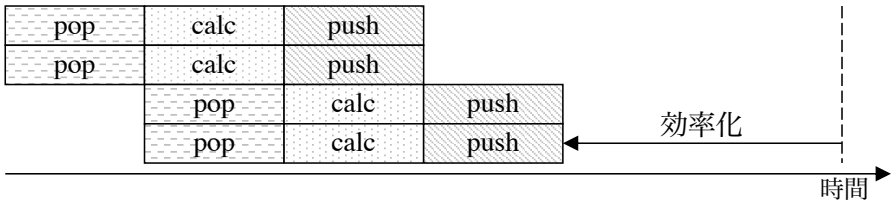
2.5　グラフ処理を行う FPGA の設計

グラフ処理を FPGA で行いたいが、一般のプログラマが FPGA 向けグラフ処理プログラムを記述するのは難しい。なぜなら、FPGA のプログラミングにはハードウェア的な知識が多分に要求されるからである。ハードウェア記述言語（HDL）による開発では、ハードウェアに関する知識が要求される。例えば HDL では、コードの記述を少し変えるだけで出力される回路が全く異なってしまうことがあるため、ハードウェアを常に考慮し、記述対象のブロックがどのような回路なのか特定しないといけない [20]。

FPGA の回路設計をする際、メモリアクセスの並列化を行うのは重要である。FPGA では演算回路を複数用意することによって、演算を並列に行い全体の処理を効率的に行うことができる。だが、演算前にメモリアクセスが必要になる場合、メモリアクセスがボトルネックとなる場合がある。図 2.9 (a) はメモリアクセスと演算がパイプライン化されたときの時系列を示している。横軸は時間であり、pop, push はメモリアクセス、calc は演算のステージを表す。



(a) パイプライン化のみ



(b) パイプライン化 + 並列メモリアクセス

図 2.9: メモリアクセスの並列化

この図が示すように、メモリアクセスを並列に行うことができないと、演算が並列化されていたとしてもその恩恵を最大限受けることができない。図 2.9 (b) のように、メモリアクセス並列度を上げることができればその問題は解決する。

グラフ処理は複雑なアルゴリズムであるため、高位合成を用いた抽象度の高い設計を行うことで、開発者はアルゴリズムの実装に集中することができる。しかし高位合成を用いた場合、メモリアクセス並列度を上げることは簡単ではない。高位合成プログラム中のある程度の大きさを持つ配列は、配列 1 つにつき 1 つの BRAM などのメモリブロックとして合成される。しかし、通常 BRAM のアクセスポートは高々 2 つであるので、プログラマの指示なしには演算の並列度に合わせてメモリを複数用意したり、オブジェクトの大きさに合わせてバンド幅を拡張したりすることはできない。また、全く同じ中身を持つ配列を 2 つ用意すると、アクセス並列度を向上させることはできても、FPGA の貴重なメモリ資源を浪費することになってしまう。この問題を解決するために、演算を工夫してメモリアクセス順序をわかりやすくし、配列を機械的に分割して複数の BRAM に配置する方法がとられるが、これは直感的なアルゴリズムの記述をするプログラマにとって障壁となる。

研究レベルでは高位合成を用いたグラフ処理システムでメモリアクセスも改善されているものがいくつか存在する。まず 1 つ目に ThunderGP[25] を紹介する。ThunderGP は、Scatter-Gather-Apply モデルによって書かれたグラフ処理アルゴリズムを FPGA で並列処理するためのフレームワークである。ユーザーは Scatter、Gather、Apply 関数をそれぞれ高級言語によって記述し、グラフデータと合わせてこのシステムの入力となる。各関数は高位合成によって演算回路となり、フレームワークが用意したアクセラレータのテンプレート回路上に複数配置される。外部メモリに置かれたグラフデータは演算のためにオンチップメモリに分割されながらキャッシュされる。この時の分割の仕方や、分割されたデータをどのようなスケ

```
1  struct Node {
2      int value;
3      int first, second;
4  };
5
6  int bfs(Node* graph){
7      int max = 0;
8      queue<int> q;
9      q.push(0);
10     while (!q.empty()) {
11         Node n = graph[q.front()];
12         q.pop();
13         if (max < n.value) max = n.value;
14         if (n.first != -1) q.push(n.first);
15         if (n.second!= -1) q.push(n.second);
16     }
17     return max;
18 }
```

図 2.10: 簡潔かつ直感的に記述された BFS プログラム

ジュールで処理していくかなども自動に決定される。このフレームワークは他のモデルを対象にしておらず、一般のグラフ処理アルゴリズムを Scatter-Gather-Apply モデルであらわすのには労力を要する。

2 つ目に SPLAG[26] を紹介する。SPLAG はダイクストラ法を行う FPGA ベースのアクセラレータである。ユーザーが用意するものはグラフデータのみである。このアクセラレータは以下の 3 種類の回路を内蔵する。

- 粗視化優先度付きキュー。近い優先度を持つ頂点を同じ優先度として扱うことで、回路のスループットを向上する。
- 頂点キャッシュ。キューへのアクセスを行う。頂点データをキャッシュすることで、レイテンシの大きい外部メモリへのアクセスを減らす。
- 更新値演算回路。入力として、キューからポップした主頂点とその隣接頂点リストを頂点キャッシュから受け取り、外部メモリから主頂点から伸びる辺のリストを受け取る。出力として、隣接頂点の更新値を頂点キャッシュに渡す。

各回路は、外部メモリのデータを FPGA のオンチップメモリにバッファリングすることにより、メモリアクセスのレイテンシを減らし、全体の処理のスループットを向上している。この手法はダイクストラ法以外を対象としていない。

一方これらのシステムでは、プログラマがアルゴリズムを直感的に記述することができるとは言えない。直感的なアルゴリズムの記述例として 2 分木に対する幅優先探索でのコードを

図 2.10 を示す。この例のプログラムは、幅優先探索によってグラフ中の各頂点が持つ値から最大の値を求める。グラフは隣接リストの形で表現されており、value は頂点が持つ値、first、second は子頂点のインデックスを表す。first、second に対応する子が存在しないことを-1 で表す。また、queue には将来探索すべき頂点のインデックスが追加される。

我々の知る限り、既存の手法ではこのような理想的な記述を実現することは非常に困難である。それは、メモリアクセスの並列性を取り出すのが難しいからである。並列グラフ処理の既存研究では、グラフアルゴリズムを隣接行列の行列積演算の形で記述させ行列積演算を並列に実行するものや、 Scatter-Gather モデルのような並列アルゴリズムの形で記述させるものが一般的である。これらの設計では、ユーザーはプログラムを非直感的な形に大きく書き換える必要があり、図 2.10 のような簡潔な記述をそのまま入力することはできない。

2.6 マルチポートメモリ

FPGA 上の複数の演算回路から発せられる複数のメモリ読み取り・メモリ書き込みを同時に処理するためのメモリ（マルチポートメモリ）が存在する [27]。従来には図 2.11 に示す 4 つの方法がとられており、それぞれの方法に利点と欠点があって用途に合わせて用いられている。

図 2.11a
: 読み出しポートの数だけメモリ数を増やしたマルチポートメモリ。一度の書き込みをすべてのメモリに反映する必要があるため、複数の書き込みポートを用意することはできない。

図 2.11b
: メモリを複数用意した（バンキング）マルチポートメモリ。ポート間でデータを共有することができない。

図 2.11c
: M 個の書き込みポートと N 個の読み取りポートと D 個のメモリを持つマルチポートメモリ。書き込む対象となるメモリを選択するために M:1 のマルチプレクサが D 個、読み込む対象を選択する D:1 のマルチプレクサが N 個必要になる。メモリ数、アドレス長、データサイズが増えた際、マルチプレクサの実装コストも増加してしまう。

図 2.11d
: メモリクロックを外部回路クロックの N 倍の周波数にすることで、N 個の読み書きポートが存在するかのようにみせる（マルチポンピング）マルチポートメモリ。読み書きの順序のセマンティクスを慎重に決定する必要がある。またポート数を増やすほど、外部回路の動作周波数が低下してしまう。

また、新しいアプローチでのマルチポートメモリの実装を提案した論文 [27] がある。この論文の提案マルチポートメモリは、マトリクス状にならんだメモリバンクと LVT (Live Value Table) の二つから構成されている。M 個の書き込みポートと N 個の読み取りポートに対し

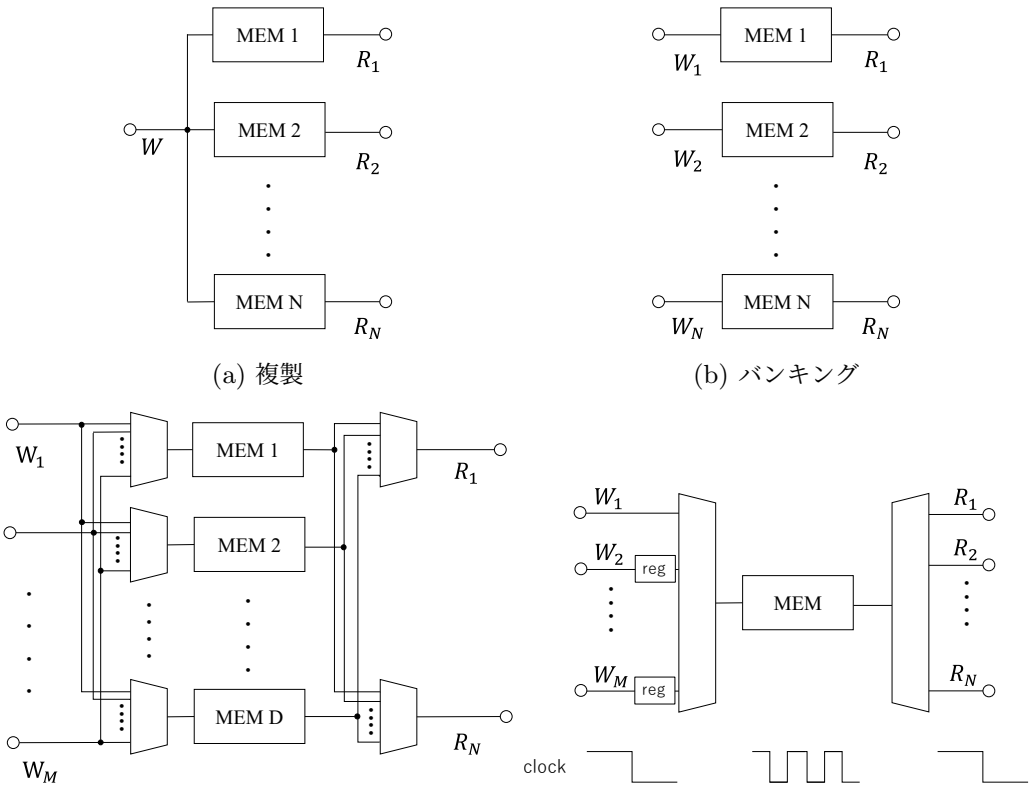


図 2.11: マルチポートメモリの実装法

て MN 個のメモリバンクが存在し、各読み書きポートの組み合わせに対応してメモリバンクが 1 つずつ存在する。LVT は各メモリ位置に対して、最新のデータがどのメモリバンクに書き込まれているか保持するためのテーブルである。書き込みの際、書き込みポートにつながるすべてのメモリにデータを書き込むと共に、LVT 上の書き込むアドレスに対応する位置に、どの書き込みポートから書き込んだのかを保持しておく。読み込みの際、読み込みポートにつながるすべてのメモリのデータを取得したのち、LVT に保持された書き込みポートに対応しているデータをマルチプレクサを用いて出力する。

第 3 章

提案フレームワーク

本研究では頂点アクセスの並列化を組み込んだ、グラフ処理用の FPGA 回路設計を支援する HLS フレームワークを提案する。このフレームワークはユーザーが簡潔にグラフ処理アルゴリズムを記述するための DSL を実際の FPGA で活用するためのものである。また、並列アクセスの実現のためのメモリコントローラも提供する。フレームワークの概要について 3.1 節で、提案 DSL の詳細について 3.2 節で、メモリコントローラの設計について 3.3 節で述べる。

3.1 概要

図 3.1 に本フレームワークの概略図を示す。このフレームワークは DSL によって記述されたグラフ処理を行う DSL プログラム、各頂点データの配置戦略、グラフデータの 3 つを入力として受け取り、頂点情報に並列アクセスする FPGA 回路を生成する。それぞれの入力と変換過程の概要について説明する。

本フレームワークは、特定の形のプログラムではプログラム中のデータのメモリ配置を動かしてもソースコードが大きく変化しないことを利用し、メモリアクセスの並列化と簡潔な記述

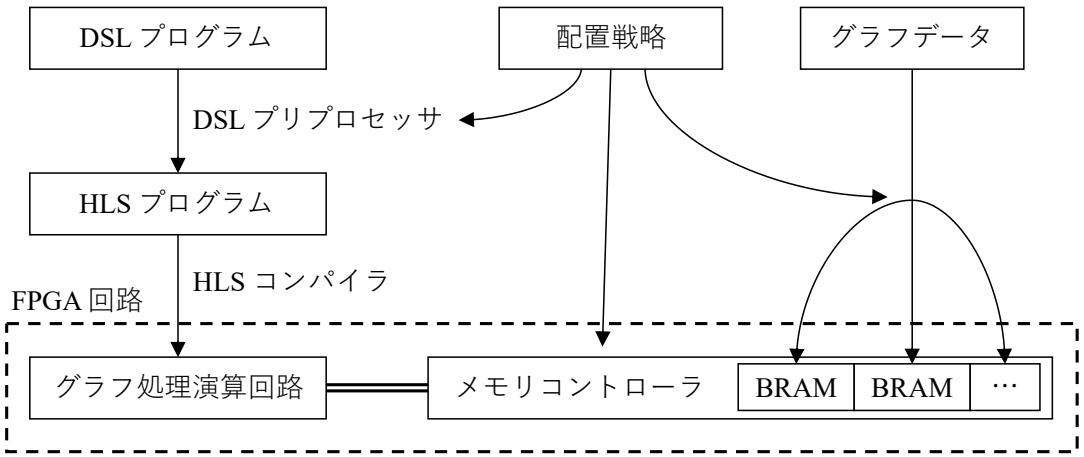


図 3.1: フレームワークの概要

を両立する。提案手法では隣接リストで表現されたグラフを、各頂点をオブジェクトとして複数の BRAM に分散させる。頂点データをどう分散配置するかの決め方を配置戦略と呼ぶ。配置戦略によってグラフ処理の性能は変化するが、配置を変更してもプログラムに大きな変更は必要ないため、プリプロセッサにおける変換で吸収することができる。

グラフデータはグラフ処理の入力であるグラフを表すデータである。グラフは頂点ごとに分散して BRAM に配置するため、隣接リストの形式とする。配置戦略によってグラフデータは分割され、複数の BRAM に配置される。グラフ処理演算回路から同時に複数の頂点インデックスを指定して BRAM 内に配置された頂点データを得たい。そのためには、並列アクセスを実現することと、どの BRAM のどの番地に目的の頂点データが配置されているか管理することが必要である。それらを実現するためにメモリコントローラを作成した。配置戦略によって異なるメモリコントローラが必要であるため、プログラマは用意された配置戦略から一つを選択して入力する。現在実装が完了しているメモリコントローラは一種類のみであるため、配置戦略はインデックスの偶奇によって別々の BRAM に配置するものしか選ぶことはできない。

DSL プログラムはどんなグラフ処理を行うのかを我々が用意した DSL を用いて記述したものである。フレームワークは DSL プログラムをプリプロセスによって HLS プログラムに変換し、さらに既存の HLS コンパイラによって回路に合成する。提案 DSL はグラフデータの配置について意識せずにプログラムを記述できるように設計した。グラフを隣接リストの形で表現しなければならない制約はあるものの、図 2.10 のような簡潔な記述に対して DSL の多少の記述を増やすことで動作させることができる。

3.2　DSL プリプロセッサ

提案する DSL の目的はグラフ処理を HLS 上で簡潔に記述可能にすることである。そのためこの DSL は、ユーザに並列アクセスを意識させないように設計されるべきである。また、メモリコントローラを介したメモリアクセスを行う際、ユーザがメモリコントローラのハードウェアについてできる限り意識をしなくて済むような設計を目指す。この節では DSL プログラムを HLS プログラムに変換するプリプロセッサと、その実装について述べる。

3.2.1　プリプロセッサによる変換

プリプロセッサによる変換により、ユーザの記述するシングルスレッドで動作することを目的としたプログラムから、FPGA 上で並列で動作し、特に複数の頂点に同時にアクセスできることを目的としたプログラムを出力する。この変換は次の 2 ステップで行う。

1. コードブロックの多重化
2. BRAM へのアクセスをメモリコントローラを使用する形に変換

本研究ではこれらの変換を行うための DSL を設計した。

本 DSL ではアルゴリズムを以下の 3 つのブロックに分けて記述する。

```
1    int max = 0;
2    queue.push(0);
3    while (!queue.empty()) {
4        Node n = queue.pop();
5        if (n.value > max) max = n.value;
6        if (n.first != -1) queue.push(n.first);
7        if (n.second != -1) queue.push(n.second);
8    }
```

図 3.2: DSL に関連する部分を省いた BFS プログラム

1. メモリコントローラに頂点アクセスのリクエストを送る部分
2. メモリコントローラのレスポンス（頂点データ）を受け取る部分
3. それ以外の計算処理を行う部分

これは我々が利用した HLS コンパイラがハードウェア合成を行う際、メモリコントローラとの通信処理とその他の計算を合わせて並列化などの最適化を行うことができないからである。そのため 3 つのブロックそれぞれについて多重化とメモリコントローラ対応を行う。ところで、グラフ処理アルゴリズムにおける頂点へのメモリアクセスの仕方にはいくつか種類がある。そのうち本 DSL ではメモリコントローラのインターフェースとして、FIFO インターフェースと Sequential インターフェースの 2 種類を提供する。FIFO インターフェースでは、リクエストが FIFO の順番で処理されてレスポンスとして pop される。Sequential インターフェースでは、すべての頂点データが頂点インデックスの順番でレスポンスに渡される。

本節では、2 分木に対する幅優先探索（BFS）により頂点の持つ値の最大値を求めるアルゴリズムを 2 並列に実行する DSL を例にとって、このステップについて説明する。なお、BFS では FIFO インターフェースを利用する。

図 3.2 に、DSL に関連する部分を省いたベースとなる BFS プログラムを示す。ここで `queue` は、頂点のインデックスを `push` すると頂点要素が `pop` されるような、配列アクセス的機能を兼ね備えたものを考えている。また、`n.value` は頂点 `n` の持つ値であり、`n.first` と `n.second` は頂点 `n` の子頂点のインデックスである。while ループ内で幅優先探索の順番で頂点が探索され、ループ終了時に変数 `max` は目的の最大値を持つ。

図 3.3 に図 3.2 で示したプログラムを DSL で記述したものを示す。このプログラムがユーザが記述する本フレームワークの入力となるプログラムである。2~9 行目は、最終的に出力される HLS プログラムが正しく動作するのに必要になる、いわゆる “お約束” の部分となっている。11、12 行目は図 3.2 における 1、2 行目と対応している。特に 12 行目のように、本 DSL では元となるプログラムを 1 つずつマクロ関数で囲うことで記述する形式をとる。13~16 行目は while ループの 1 ループ目を取り出したものとなっている。第 1 ループは `queue` の中の要素が 1 つしかないため、並列に実行することができない。そのため特別扱いする必要があり、このような形をとった。18~24 行目は図 3.2 における 3~8 行目と対応している。また、

```
1  void binary_tree_bfs(
2      hls::stream<Node>& graph_fifo0,
3      hls::stream<Node>& graph_fifo1,
4      hls::stream<ap_uint<8>>& addr_fifo0,
5      hls::stream<ap_uint<8>>& addr_fifo1,
6      hls::stream<ap_int<32>>& out_r) {
7      Node n0, n1;
8      graph_fifo0.read_nb(n0);
9      graph_fifo1.read_nb(n1);
10
11     ap_int<32> max = 0;
12     OUTPUT_BLOCK(n, 2, QUEUE_PUSH(0););
13     INPUT_BLOCK(n, 2, QUEUE_POP(n););
14     CALCULATION_BLOCK_1(n, if (max < n.value) max = n.value;);
15     FIRST_OUTPUT_BLOCK_1(n, if (n.first != -1) QUEUE_PUSH(n.first););
16     FIRST_OUTPUT_BLOCK_2(n, if (n.second != -1) QUEUE_PUSH(n.second););
17
18     while (true) {
19         INPUT_BLOCK(n, 2, QUEUE_POP(n););
20         CALCULATION_BLOCK(n, 2, if (max < n.value) max = n.value;);
21         out_r << max;
22         OUTPUT_BLOCK(n, 2, if (n.first != -1) QUEUE_PUSH(n.first););
23         OUTPUT_BLOCK(n, 2, if (n.second != -1) QUEUE_PUSH(n.second););
24     }
25 }
```

図 3.3: 本フレームワークの DSL により記述された BFS プログラム

図 3.2 と 図 3.3 では while ループの条件が `!queue.empty()` から `true` に変わっている。これは、回路全体としてはメモリコントローラの empty 信号が 1 になった時に終了することにしたため、 `!queue.empty()` はあってもなくても同じだからである。

図 3.4 に図 3.3 の while ループの中を二重化したものを示す。図 3.3 では 1 つだった `max` を更新する計算と Queue に対する操作が図 3.4 では 2 倍に増えている。このようにコードブロックを二重化することで後段の HLS コンパイラによって並列にスケジュールされることが期待できる。メモリアクセスを並列に実行するには頂点情報を読み出すコードブロックと読み出された頂点情報に依存するコードブロックを全て二重化する必要がある。これを実現するためのマクロの実装方法については 3.2.2 節で説明する。

図 3.5 に図 3.4 中の BRAM へのアクセス（コード上では Queue の pop 操作と push 操作）を 3.3 節で後述するメモリコントローラを使用するように変換したプログラムを示す。`addr_fifo` と `graph_fifo` は、メモリコントローラの入出力にある FIFO を表す変数である。

```
1    while (true) {
2        QUEUE_POP(n0);
3        QUEUE_POP(n1);
4        if (max < n0.value) max = n0.value;
5        if (max < n1.value) max = n1.value;
6        out_r << max;
7        if (n0.first != -1) QUEUE_PUSH(n0.first);
8        if (n1.first != -1) QUEUE_PUSH(n1.first);
9        if (n0.second != -1) QUEUE_PUSH(n0.second);
10       if (n1.second != -1) QUEUE_PUSH(n1.second);
11   }
```

図 3.4: while ループ内を二重化したプログラム

```
1    while (true) {
2        graph_fifo0 >> n0;
3        graph_fifo1 >> n1;
4        if (max < n0.value) max = n0.value;
5        if (max < n1.value) max = n1.value;
6        out_r << max;
7        if (n0.first != -1) addr_fifo0 << n0.first;
8        if (n1.first != -1) addr_fifo1 << n1.first;
9        if (n0.second!= -1) addr_fifo0 << n0.second;
10       if (n1.second!= -1) addr_fifo1 << n1.second;
11   }
```

図 3.5: Queue の操作をハードウェアに対応させたプログラム

addr_fifo に頂点のインデックスを書き込むとメモリコントローラはその値を読み出し、対応する頂点データを graph_fifo に書き込む。メモリコントローラはリクエストが同じ BRAM に集中していない限り並列に全てのリクエストを適切な BRAM に割り当てるように設計してあるため、二重化したコードブロックが期待通りに並列にスケジュールされればメモリアクセスを並列に処理することができる。

3.2.2 マクロの実装

DSL は C++ 言語のマクロを利用した Embedded DSL (EDSL) である。EDSL とは、ホストとなるプログラム処理系（ここでは HLS）の上に、ライブラリやマクロによって実装された DSL である。本実装でマクロによる EDSL にする理由は、実装が簡単かつ提案の実現に十分であると考えられるためである。

```
#define OUTPUT_BLOCK_1(V, CODE) { auto V = V ## 0; \
    auto& addr_fifo = addr_fifo##0; CODE }
#define OUTPUT_BLOCK_2(V, CODE) OUTPUT_BLOCK_1(V, CODE) \
    { auto V = V ## 1; auto& addr_fifo = addr_fifo##1; CODE }
#define OUTPUT_BLOCK(V, N, CODE) do { OUTPUT_BLOCK_##N(V, CODE) } \
    while(0)
```

図 3.6: OUTPUT_BLOCK の実装

```
#define QUEUE_PUSH(V) addr_fifo << V
```

図 3.7: QUEUE_PUSH の実装

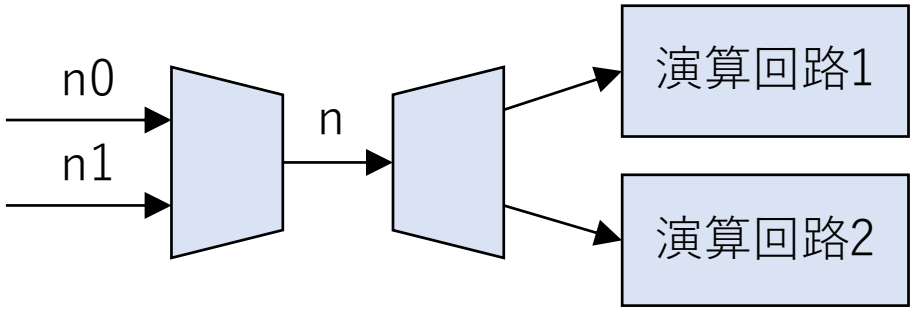


図 3.8: 変数名の置き換えが HLS ツールによって最適化されなかった回路

OUTPUT_BLOCK の実装

本小節では図 3.3 における OUTPUT_BLOCK について、その機能と実装を説明する。

OUTPUT_BLOCK は Queue への push に対応する部分のプログラムを多重化するためのマクロである。第一引数の n は、プログラム中で頂点データを格納するための変数の prefix である。この場合、n0、n1··· のように変数を用意しておくものとする。第二引数にはメモリコントローラ内の BRAM の数を指定する。これは HLS 関数内で行う演算の並列処理の数と一致する。第三引数に、並列化したい処理の内容を記述する。図 3.3 の場合、n の子頂点のインデックスが -1（無効であることを示す）でないときに Queue に push するようなプログラムを指定している。

図 3.6 と図 3.7 にそれぞれ OUTPUT_BLOCK と QUEUE_PUSH の実装を示す。OUTPUT_BLOCK##N のようにトークン結合を用いることによってマクロ関数を再帰的に展開することができる。また、OUTPUT_BLOCK の 1 文目は auto n = n0; のように展開され、CODE 内の n の指すものを変更することが可能になっている。この場合、n や n0、n1 のような変数は高位合成ツールは配線名とみなされるが、図 3.8 のようにプログラムをそのまま表すような無駄なマルチプレクサ・デマルチプレクサが挿入されることはない。その代わり最適化によって図 3.9 のようにそ

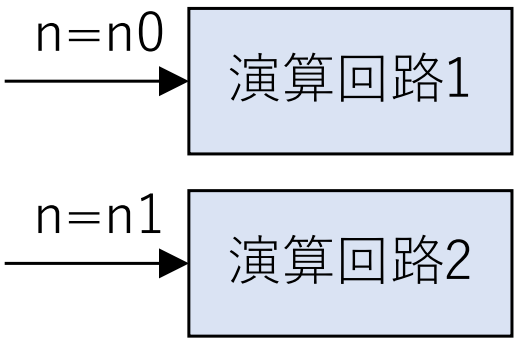


図 3.9: 最適化によって変数名の置き換えが無視された回路

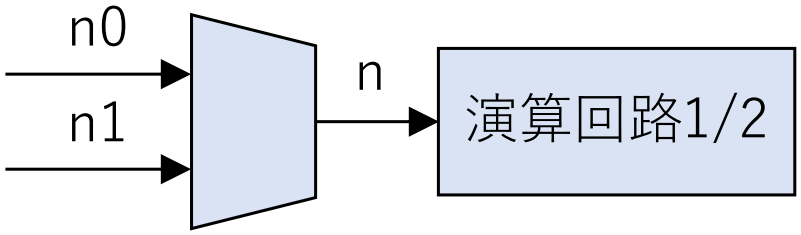


図 3.10: 演算回路が似ているため最適化によってまとめられた回路

れぞれ独立した回路が生成されるか、あるいは図 3.10 のように演算回路が似ている場合は統合されたものが生成される。ただし、図 3.10 の場合並列演算ができないため、この場合は基本的に図 3.9 のような回路が生成されると考えてよい。

図 3.11 にて、図 3.6 と図 3.7 のマクロによる展開前後を比較した。展開前が図 3.3 の 19 行目、展開後が図 3.5 の 7、8 行目に対応している。DSL プログラムの while ループ中に登場したプログラムが、適切に二重化とメモリコントローラへの対応がなされていることが確認できる。

Sequential インターフェース

本小節では Sequential インターフェースにおけるメモリアクセスの仕方と関連するマクロについて説明する。

図 3.12 は Sequential インターフェースを利用する DSL の一部を切り抜いたものである。`READ_SEQ` マクロは、プレフィックス `n` をもつ頂点変数にインデックス順にデータを格納することを表している。第 2 引数はメモリコントローラが持つ BRAM の数、第 3 引数はそのループで扱う頂点インデックスを指定する。このメモリアクセスはメモリコントローラが BRAM を 2 つもつ場合、図 3.13 に示すプログラムに変換される。

3.3 メモリコントローラ

本研究で作成したメモリコントローラについて説明する。メモリコントローラはグラフ処理カーネルから入力されたリクエストに対して適切な頂点情報を BRAM から読み出し、出力

```
// before expansion
OUTPUT_BLOCK(n, 2, if (n.first != -1) QUEUE_PUSH(n.first););

// after expansion
do {
    {
        auto n = n0;
        auto& addr_fifo = addr_fifo0;
        if (n.first != -1) addr_fifo << n.first;
    }
    {
        auto n = n1;
        auto& addr_fifo = addr_fifo1;
        if (n.first != -1) addr_fifo << n.first;
    }
} while(0);
```

図 3.11: OUTPUT_BLOCK による展開前後の比較

```
    ap_uint<8> i = 0;
    while (i < GRAPH_SIZE) {
        Node n0, n1;
        READ_SEQ(n, 2, i);
        CALCULATION_BLOCK(n, 2,
            ...
            i++;
        );
    }
```

図 3.12: Sequential インターフェースを利用する DSL

する。メモリコントローラは複数の入力ポートを持ち、複数のリクエストを同時に入力することができる。複数のリクエストが同時に入力されたとしても、対応する頂点情報が別々の BRAM に格納されているならば並列に読み出し、入力ポートに対応する出力ポートから出力する。もし同時に入力されたリクエストが同じ BRAM に格納されていた場合は、頂点情報の読み出しは順番に行い、頂点情報の読み出しが終わったリクエストから順に出力に書き込む。そのためメモリコントローラの応答時間は変動する場合があり、入出力は FIFO を介して行う。

メモリコントローラを配置戦略に応じた状態遷移を持つ順序回路として実装した。例えば 2 つの BRAM を持つメモリコントローラの配置戦略を「インデックスの偶奇によってデータがどちらの BRAM に配置されるか決まる」とする。この場合、アクセスの並列度は高々 2 であ

```
1    addr_fifo0 << i;
2    addr_fifo1 << i + 1;
3    while (graph_fifo0.empty() || graph_fifo1.empty());
4    graph_fifo0 >> n0;
5    graph_fifo1 >> n1;
```

図 3.13: Sequential インターフェースを利用する DSL

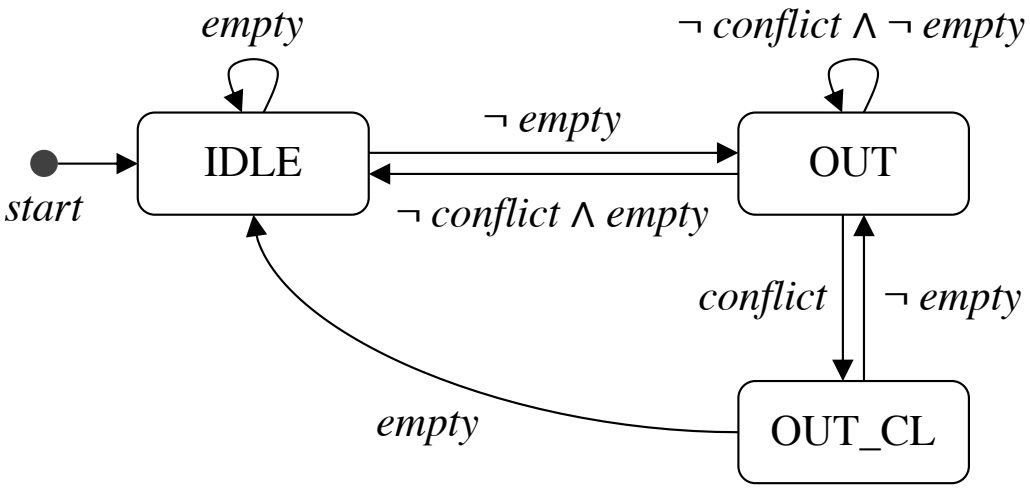


図 3.14: 有限状態機械の状態遷移図

るので、メモリコントローラの入出力ポートは 2 個ずつ持つ。入力ポートに偶数インデックスと奇数インデックスが同時に指定された場合、それらは別の BRAM にデータが配置されているため、並列に読み出すことができる。しかし、入力が偶数インデックス 2 つあるいは奇数インデックス 2 つだった場合、それらは同じ BRAM に配置されているため、並列に読み出すことができない。いずれの状態であっても可能な限り早くデータを出力するように、それぞれの状態を表現する有限状態機械を内蔵したメモリコントローラを実装した。

図 3.14 にメモリコントローラの状態を管理する有限状態機械の状態遷移図を示す。まず、アクセスが無い時、メモリコントローラは IDLE 状態である。2 つの入力ポートにインデックスが入力されたとき、OUT 状態に移行する。この時、入力が偶数と奇数であれば、並列に 1 クロックで処理が可能だが、偶数 + 偶数あるいは奇数 + 奇数であると、1 つの BRAM に対して 2 回アクセスする必要があるため、処理に 2 クロックかかる。このように、1 つの BRAM に対するアクセスが同時に必要とされる入力を、以降ではコンフリクト入力と呼ぶことにする。コンフリクト入力での 2 クロック目は状態を区別しないといけないため、そのような場合は OUT_CL 状態に移行することとする。

図 3.15 に状態遷移を verilog で実装したコードを示す。`state` は現在の状態を保存するレジスタである。クロック毎に状態の変更が行われる。`case` 文は C 言語における `switch` 文のようなもので、`state` のもつ値に応じて条件分岐が行われる。`fifo_empty == 2'b00` は 2

```
1    reg [0:1] state;
2    always @(posedge clk) begin
3        case (state)
4            IDLE:
5                if (fifo_empty == 2'b00) state <= OUT;
6                else state <= IDLE;
7            OUT:
8                if (ad0_parity == ad1_parity) state <= OUT_CL;
9                else if (fifo_empty == 2'b00) state <= OUT;
10               else state <= IDLE;
11           OUT_CL:
12               if (fifo_empty == 2'b00) state <= OUT;
13               else state <= IDLE;
14           default:
15               state <= IDLE;
16       endcase
17   end
```

図 3.15: 状態遷移を表す verilog コード

表 3.1: コンフリクト入力でない場合の OUT 状態でのマルチプレクサの選択信号

入力の偶奇	MUX			
(port1, port2)	1	2	3	4
(偶数, 奇数)	0	1	0	1
(奇数, 偶数)	1	0	1	0

つの入力ポートにインデックスが入力されていること、`ad0_parity == ad1_parity` はコンフリクト状態であることを示している。それぞれの条件に合った時、図 3.14 に示すような状態遷移が行われることをレジスタに対する代入で表している。

図 3.16 に本メモリコントローラ回路の設計概略を示す。有限状態機械により決定された状態に応じて、4 つのマルチプレクサが信号を適切に選択する。ポート 1 とポート 2 の偶奇は異なるとき、入力はコンフリクト入力ではない。BRAM1,2 がそれぞれ偶数インデックス、奇数インデックスのデータが配置されているとすると、マルチプレクサの選択信号表は表 3.1 のようになる。この表は、port1,2 の入力の偶奇に対応したマルチプレクサ 1∼ 4 の選択信号を示している。入力がコンフリクト入力の場合、OUT 状態では port1 の入力を、OUT_CL 状態では port2 の入力を処理することにした。この場合、OUT 状態でのマルチプレクサの選択信号表を表 3.2 に、OUT_CL 状態でのマルチプレクサの選択信号表を表 3.3 に示した。x はその選択信号が 0 でも 1 でも動作に影響を与えないということを示している。

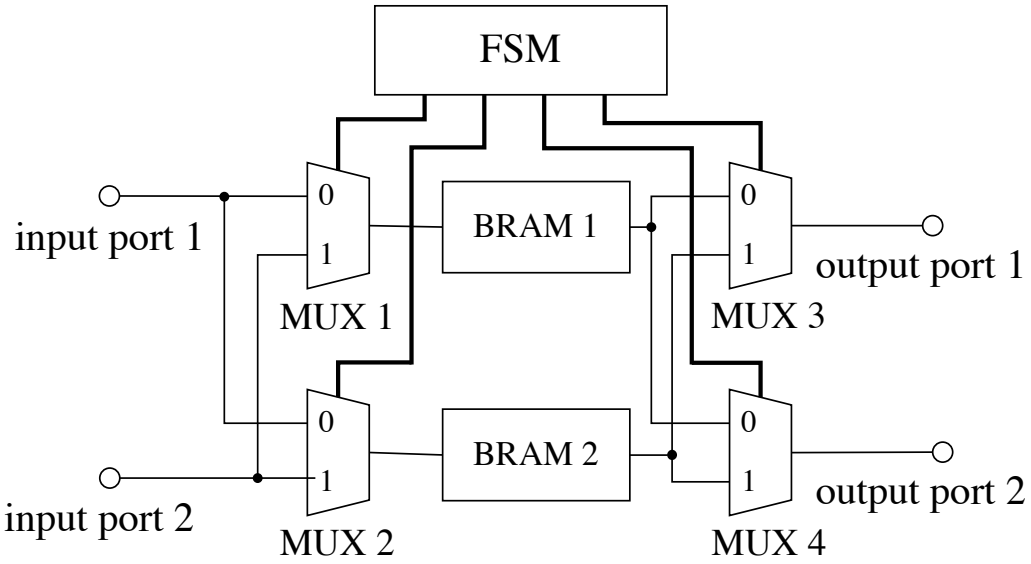


図 3.16: メモリコントローラ回路の概略

表 3.2: コンフリクト入力の場合の OUT 状態でのマルチプレクサの選択信号

入力の偶奇	MUX			
(port1, port2)	1	2	3	4
(偶数, 偶数)	0	x	0	x
(奇数, 奇数)	x	0	1	x

表 3.3: コンフリクト入力の場合の OUT_CL 状態でのマルチプレクサの選択信号

入力の偶奇	MUX			
(port1, port2)	1	2	3	4
(偶数, 偶数)	1	x	x	0
(奇数, 奇数)	x	1	x	1

第 4 章

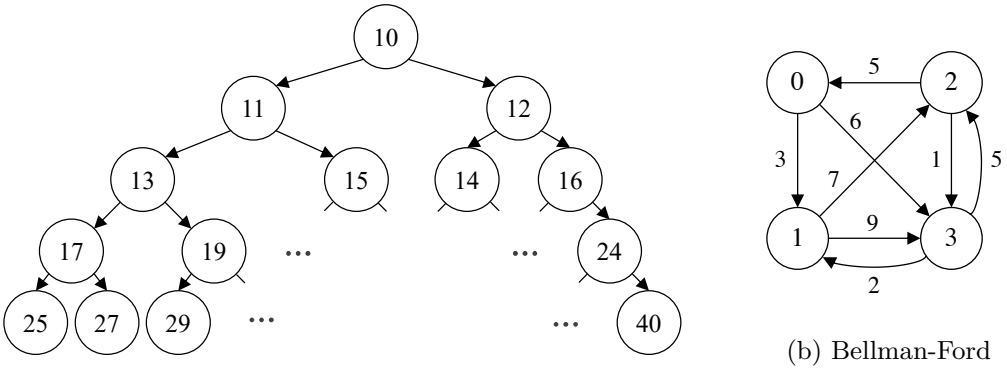
実装と評価

4.1 評価対象

二分木に対する幅優先探索と、Bellman-Ford アルゴリズムを用いて評価を行った。評価は Vivado のシミュレータ上で行う都合、小さなグラフに対して適応することにした。

幅優先探索を用いた評価では、ベンチマークプログラムは入力された二分木を走査し、各頂点が持つ値の中で最大の値を出力する。幅優先探索の入力に使用したグラフを図 4.1 (a) に示す。各頂点の数字はその頂点が持つ値である。Bellman-Ford アルゴリズムを用いた評価では、ベンチマークプログラムは二頂点間の最短経路長を出力する。Bellman-Ford アルゴリズムの入力に使用したグラフを図 4.1 (b) に示す。各辺に添えた数字はその辺の重みである。

また、DSL や並列アクセスの有無によってどのように性能が変化するかを検討する。すなわち、第一にすべての頂点データが格納されるメモリコントローラ内部の BRAM の数が 2 つある場合と、BRAM の数が 1 つの場合の処理にかかる実行サイクルを比較することで、提案フレームワークによって並列性が向上できることを確認する。第二に DSL を用いてアルゴリズムを実装した場合と、HLS C++ を直接記述して実装した場合の処理にかかる実行サイクルとプログラムを比較することで、提案 DSL を用いた際にも並列性が向上していることとプロ



(a) 2 分木幅優先探索

(b) Bellman-Ford

図 4.1: 今回の評価で用いたグラフ

グラムの記述が簡単になっているかを検討する。そのため、以下の 3 つのパターンの方法を 2 つのアルゴリズムそれぞれについて用意して評価対象とした。

1. DSL による実装・メモリコントローラ内部の BRAM は 2 つ
2. DSL を用いない実装・メモリコントローラ内部の BRAM は 2 つ
3. DSL を用いない実装・メモリコントローラ内部の BRAM は 1 つ

また実行サイクルの計測は Vivado のシミュレータ上で行った。シミュレーションのための具体的な実装については 4.2 節で述べる。

4.2 シミュレーションの実装

フレームワークから出力された回路の動作と実行サイクルの計測は Vivado のシミュレータ上でのみ行った。表 4.1 に今回のシミュレーションに用いた環境を示す。なお、HLS コンパイラとして Vitis HLS を用い、生成された RTL ファイルを Vivado で読み込むことでシミュレーションを行った。

図 4.2 はシミュレーションのために用意した回路の論理図である。Vitis HLS によりコンパイルされた回路ブロックと、メモリコントローラとの配線は Vivado 上で手動で行った。図 4.2 の緑色で示した配線がそれに当たる。グラフ処理演算回路とメモリコントローラは FIFO を介して接続されている。また、これらの回路全体に対する入力としてオレンジ色に示した配線を用意した。それぞれについて説明する。まず演算回路制御信号は、提案 DLS から出力される演算回路を制御するための信号である。`ap_start` は演算を開始するための合図をする信号であり、`reset` は演算回路の状態にかかわらずリセットするための信号である。クロック信号は演算回路やメモリコントローラ内の順序回路で用いられる外部クロックを与えるための信号である。頂点データ書き込みポートはメモリコントローラ内の BRAM に直接接続されており、演算回路が開始する前にグラフ頂点データを書き込むために用いられる。これらの信号はシミュレーションの testbench コードによって制御されている。

またこれらの信号の他にも、シミュレーションで内部信号を監視するための配線をしている。FPGA のシミュレーションを用いたデバッグでは `gdb` のような CPU 上の処理系で用いるデバッガとは異なり、プログラム中の変数や分岐などを追いかけることができないため、回路の内部信号を監視することで行われる。このとき外部から監視し waveform 上で見ることを回路とシミュレーションで明示しないといけないので、そのための配線を行った。

testbench コードを図 4.3 に示す。この testbench は二分木に対する幅優先探索を 2 つの BRAM を有するメモリコントローラを用いて行うシミュレーションのために記述した。これについて説明する。

1 行目、11-13 行目

クロックを与えるための記述である。1 行目でシミュレーションのタイムスケールを指定している。左の `1ns` はシミュレーションで用いる単位時間（`#1` により 1 単位時間経

表 4.1: シミュレータの動作環境とシミュレートされる環境

Windows	バージョン	Windows 11 Education 21H2
Vitis HLS	バージョン	2020.2
	Part Selection	xc7z020clg400-1
Vivado	バージョン	2020.2
	Project Part	xc7z020clg400-1

過される)、右の `10ps` はシミュレーションの丸め精度（これより小さい信号変化タイミングは四捨五入される）を表す。11-13 行目でクロック信号を生成している。5000 単位時間おきにでクロックを上げ下げしているため、クロック周期は 1 ns の 10000 倍、つまり 10 µs となる。

2-5 行目

配線などを定義している。記述が冗長なため... の部分は省略したが、図 4.2 に示した配線とデバッグ信号の配線をしている。

8-10 行目、37-39 行目

演算回路制御信号を与える部分である。シミュレーション開始時は `ap_start` は下げておき、演算開始するのに十分な頂点データが書き込まれた 12 周期分が経過したときに、`ap_start` を立ち上げることによって演算開始する。また `reset` はシミュレーション開始時に一度だけ立ち上げることで演算回路を初期化し、以降はずっと下げておく。そしてシミュレーション開始から 162 周期目でシミュレーションを終了するために、コマンド `$finish` を記述している。

15-36 行目

BRAM にグラフ頂点データを書き込む部分である。図 4.1a に示したグラフを隣接リスト形式で BRAM に書き込んでいる。偶数インデックスの頂点が BRAM0 に、奇数インデックスの頂点が BRAM1 に書き込まれるようにしている。

4.3　並列性の評価

フレームワークによってグラフ処理が並列化されているということを確認するための評価を行った。

図 4.4 に 1 つの BRAM を備えたメモリコントローラを用いて幅優先探索のシミュレーションを行った時に出力される waveform を示す。waveform とは FPGA 内部にある各デジタル信号の変化を時系列で表示するものである。横軸は時系列、縦軸は各信号である。この waveform の信号を観察することで HLS によって出力された回路が計算に使用したサイクル数を計測する。幅優先探索と Bellman-Ford とで異なる方法で計測を行なったので、それぞれについて説明する。

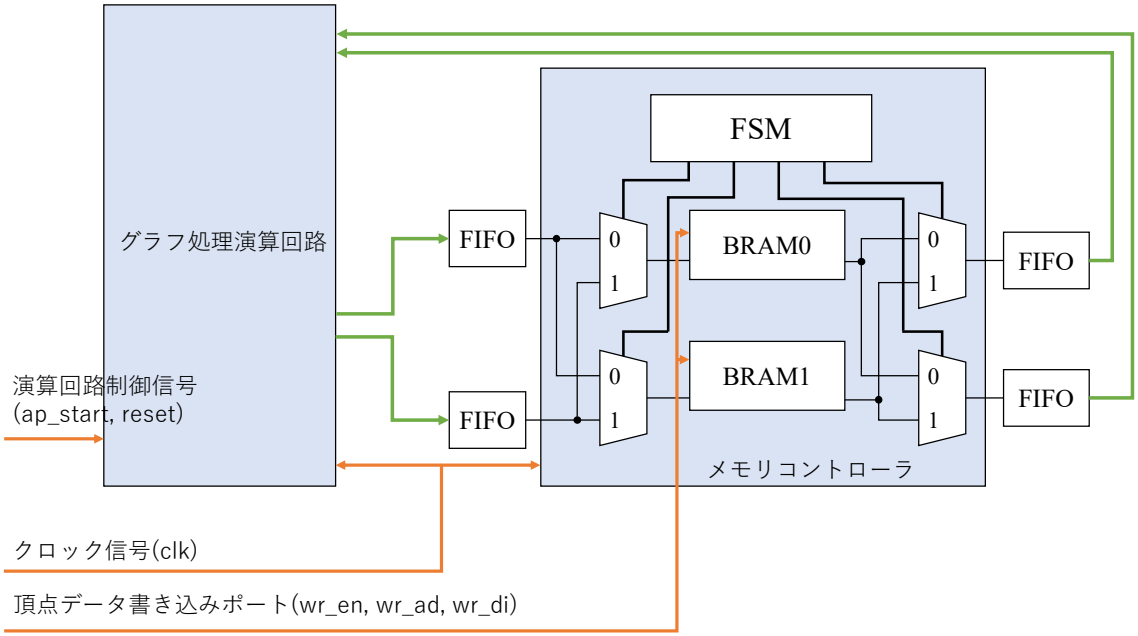


図 4.2: シミュレーション回路の論理図

二分木幅優先探索での計測方法を説明する。図 4.4 に幅優先探索によって得られた waveform を示す。計測開始は回路の動作開始のトリガーとなる `ap_start` 信号が立ち上がった時とした。計測終了は、`fifo_reader_dout` という信号から、求めたい答えが出力された時とした。この信号は HLS によって出力された回路が FIFO 経由で探索された頂点の値の最大値を出力しているものである。今回のグラフの最大値は 0x28 であり、図 4.4 の縦黄色破線のタイミングが計測終了にあたる。

次に Bellman-Ford アルゴリズムの計測方法を説明する。図 4.7 に Bellman-Ford アルゴリズムによって得られた waveform を示す。今回使用したグラフでは実行の早い段階で最短経路長が求まってしまうため、幅優先探索と同じ方法で計測することができなかった。代わりにメモリコントローラ内部の有限状態機械が特定の状態の時に 1 になる `in_state_debug_0` を出力に追加し、1 になった回数を観察することで計測を行った。一度の Bellman-Ford アルゴリズムの実行の中で何回 `in_state_debug_0` が 1 になるかはプログラムとグラフから知ることができる。`ap_start` が 1 になるタイミングと `in_state_debug_0` が初めて 1 になるタイミングの間にはズレがあるが、このズレは計算が完了してから再び `in_state_debug_0` が 1 になるまでのズレと同じ長さであるはずなので、`in_state_debug_0` が初めて 1 になった時点から 7 回目に 1 になった時点までの時間を実行時間として計測した。

図 4.4 ～ 図 4.9 に、本評価で計測したすべての waveform を示す。前述した方法でこれらの waveform から計算にかかったサイクル数を計測した。また、図 4.10 に 6 パターンで計算にかかったサイクル数の比較を示す。縦軸はサイクル数であり、横軸は計測条件を示している。

提案メモリコントローラで BRAM を 2 つ用いた場合、1 つ用いたときに比べて、2 分木幅優先探索で約 68%、Bellman-Ford で約 63% のサイクル数で処理することができた。BRAM

```
1  `timescale 1ns / 10ps
2  module binary_tree_bfs_2bram_tb ();
3      integer i, j;
4      ...
5      binary_tree_bfs_2bram_wrapper db(...);
6
7  // sys
8      initial begin
9          ap_start <= 0; reset <= 1; #10000 reset <= 0;
10     end
11     always begin
12         clk = 0; #5000 clk = 1; #5000;
13     end
14 // write
15     initial begin
16         wr_en_0 = 1; wr_en_1 = 1;
17         for (i = 0; i < 7; i = i + 1) begin
18             wr_ad_0 <= i;
19             wr_di_0 <= (i*4+2) << 40 | (i*4+1) << 32 | i*2+10;
20             wr_ad_1 <= i;
21             wr_di_1 <= (i*4+4) << 40 | (i*4+3) << 32 | (i*2+1)+10;
22             #10000;
23         end
24         wr_ad_0 <= 7;
25         wr_di_0 <= (7*4+2) << 40 | (7*4+1) << 32 | 7*2+10;
26         wr_ad_1 <= 7;
27         wr_di_1 <= (-1) << 40 | (-1) << 32 | (7*2+1)+10;
28         #10000;
29         for (i = 8; i < 16; i = i + 1) begin
30             wr_ad_0 <= i;
31             wr_di_0 <= (-1) << 40 | (-1) << 32 | i*2+10;
32             wr_ad_1 <= i;
33             wr_di_1 <= (-1) << 40 | (-1) << 32 | (i*2+1)+10;
34             #10000;
35         end
36     end
37     initial begin
38         #120000 ap_start <= 1; #1500000 $finish;
39     end
40 endmodule
```

図 4.3: 二分木に対する幅優先探索を 2 つの BRAM を有するメモリコントローラを用いて行うシミュレーションのための testbench コード

アクセスが並列に行えることにより、FPGA 上の計算も並列に行えるようになっていることがわかった。理想的には半分の時間で処理できているとよいが、そうならなかった理由として次の 2 点が挙げられる。

1. HLS トップレベル関数の最初にアルゴリズムの初期化部分があり、フレームワークによる処理効率向上ができない。
2. 幅優先探索の場合、パイプライン開始直後に、メモリコントローラの出力側の FIFO が一時的に空となり、パイプラインが stall してしまうため、フレームワークによる処理効率向上ができない。

また、DSL を用いた場合と DSL を用いなかった場合、ほぼ同じサイクル数で処理が終わっている。これにより、DSL によって FPGA 上での並列計算に影響を与えていないことがわかる。

4.4 DSL の簡単さの評価

提案する DSL によるグラフ処理の書きやすさを評価するため、DSL によって記述されたグラフ処理プログラムと、他のいくつかのプログラムを比較する。

まず、並列グラフ処理アルゴリズムの一つである Scatter-Gather モデルによる記述を比較対象として、既存手法より書きやすくなっていることを確認する。なお、関連研究である ThunderGP[25] では Gather-Apply-Scatter モデルが用いられている。これは Scatter-Gater モデルを発展して、頂点の次数分布がべき乗則に従う Power-Law グラフにおいて性能改善するためのものである [28]。なお、Gather-Apply-Scatter モデルによるグラフアルゴリズムの記述について Kalavri ら [28] の論文の Algorithm 8, 10 に、Scatter-Gather モデルは Algorithm 4, 7 に記載がある。一般に Scatter-Gather モデルのほうが Gather-Apply-Scatter モデルより簡単な記述が可能である。図 4.11 に Scatter-Gather モデルにより記述された BFS プログラムを載せる。これと DSL を用いた BFS プログラム（図 3.3）を比較して簡単になっているということを説明する。提案 DSL のプログラムと異なり、Scatter-Gather モデルでの記述には以下の 2 点が必要となる。

- アルゴリズムを Scatter-Gather モデルの形に変形する。
- scatter/gather 関数をそれぞれ頂点間メッセージパッシングインターフェースを利用して記述する。

これに比較して提案 DSL のプログラムはマクロによるコードブロックへのアノテーションは必要なものの、アルゴリズムの形を変えることなく記述できる。以上より提案 DSL での記述は ThunderGP で用いられている Gather-Apply-Scatter モデルによる記述より直感的に記述できると言える。

2 つ目に、提案フレームワークによりアルゴリズムを記述するのが簡単になっていることを、プログラムの記述量を比較することで確認する。具体的には、提案メモリコントローラを用い

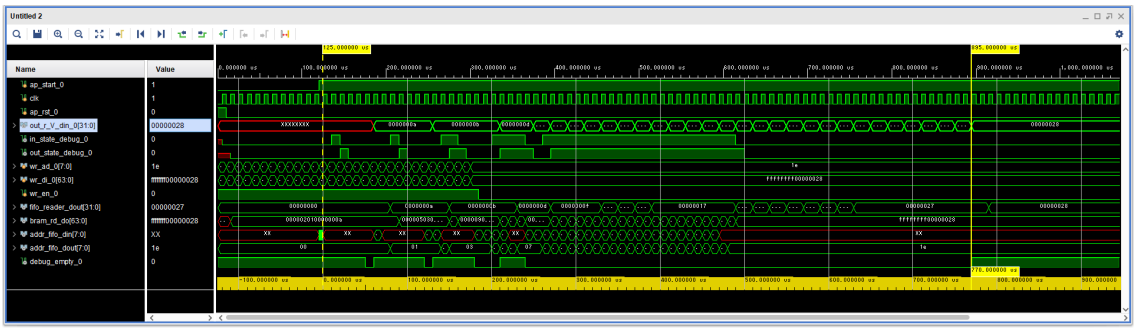


図 4.4: 幅優先探索を BRAM1 つでシミュレーションを行った時に出力される waveform

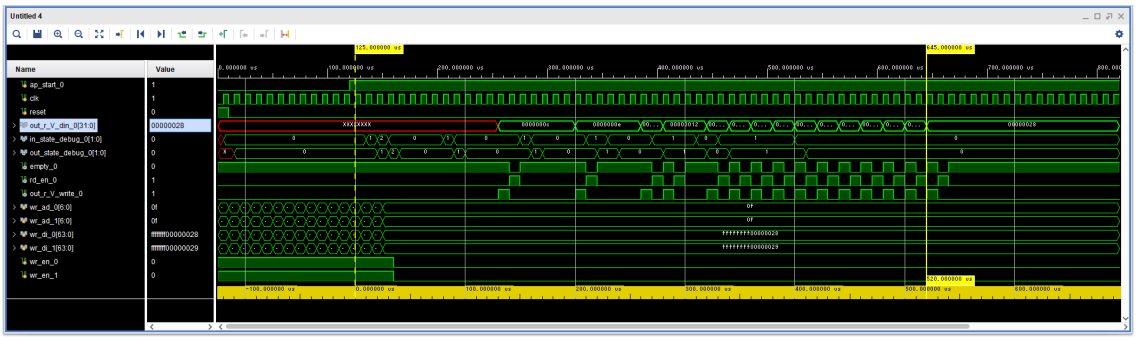


図 4.5: 幅優先探索を BRAM2 つでシミュレーションを行った時に出力される waveform

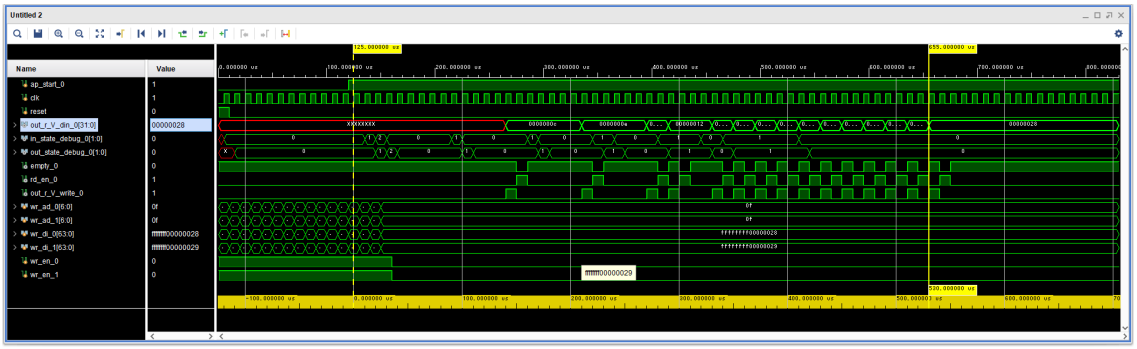


図 4.6: 幅優先探索を DSL を用いてシミュレーションを行った時に出力される waveform

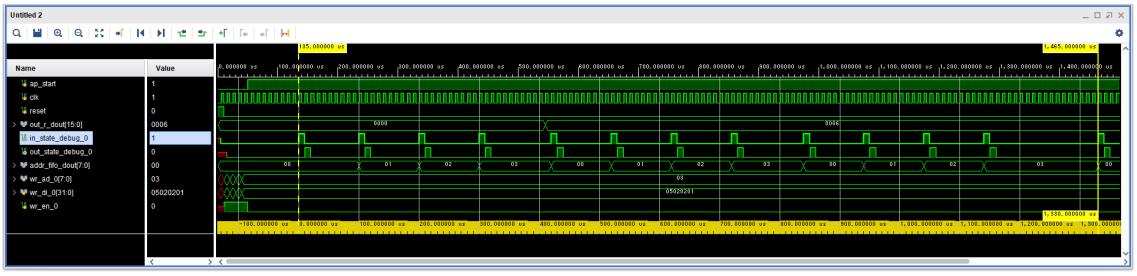


図 4.7: BellmanFord 法を BRAM1 つでシミュレーションを行った時に出力される waveform

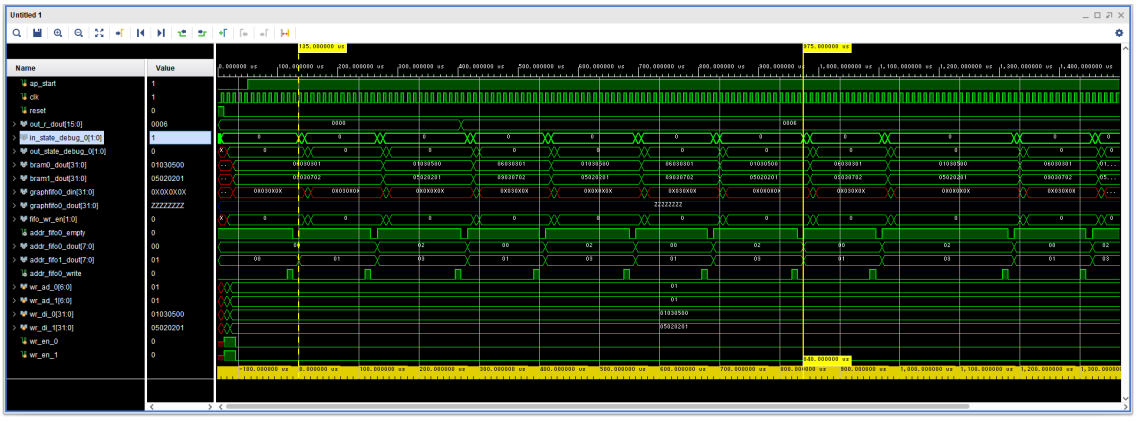


図 4.8: BellmanFord 法を BRAM2 つでシミュレーションを行った時に出力される waveform

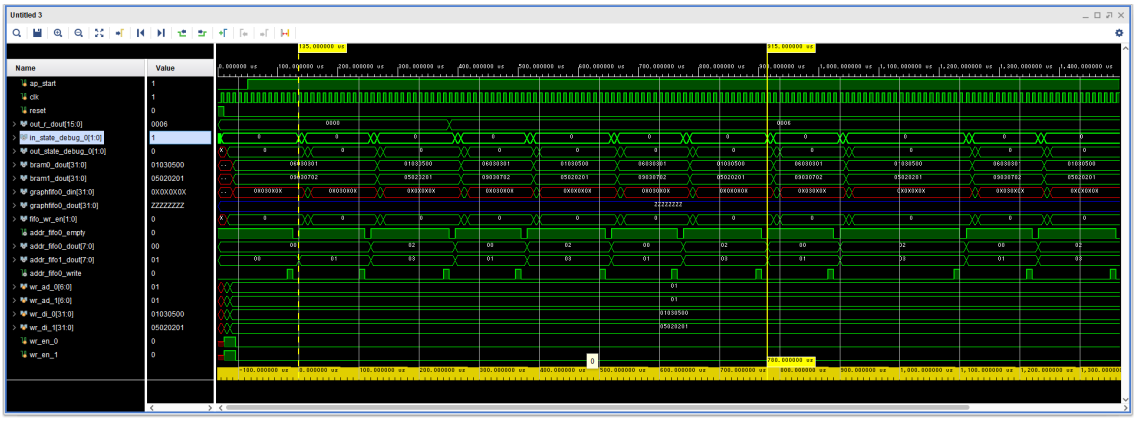


図 4.9: BellmanFord 法を DSL を用いてシミュレーションを行った時に出力される waveform

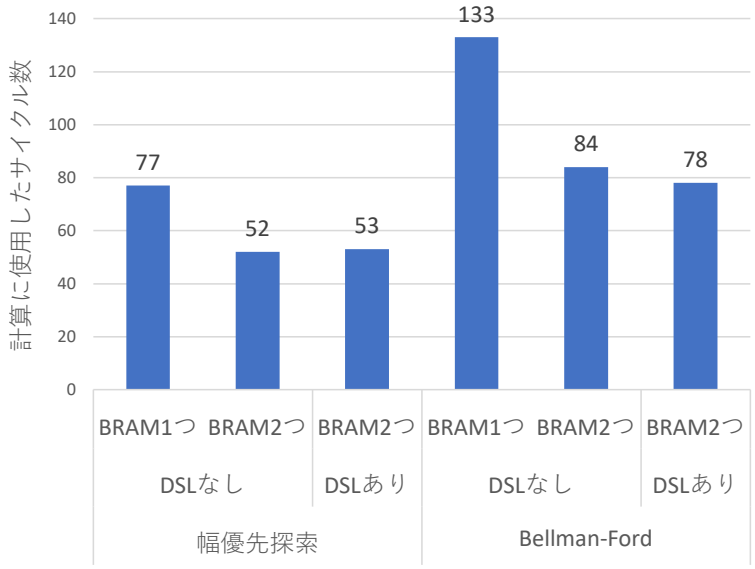


図 4.10: 並列性の評価結果

た並列計算を行う HLS C++ プログラムを提案 DSL で書き換えたとき、プログラムの行数が減っているか評価を行う。図 4.12 は図 3.3 と同等の、DSL を用いずに書かれた BFS プログラムである。DSL を用いて書くことで、24 行（図 4.12 の 7 ～ 30 行目）から 18 行（図 3.3 の 7 ～ 24 行目）に減っている。また、関数引数（図 3.3 の 2 ～ 6 行目）や初期化部分（図 3.3 の 8 ～ 9 行目）を除けば、ハードウェアを扱うために必要な関数が DSL のマクロ関数によって隠蔽されている。

Bellman-Ford 法についても同様の比較を行う。図 4.13 は DSL を用いて書かれた Bellman-Ford 法プログラムであり、図 4.14 は同様の処理を DSL を用いずに書かれたプログラムである。DSL を用いて書くことで、24 行（図 4.14 の 14 ～ 37 行目）から 18 行（図 4.13 の 14 ～ 31 行目）に減っている。また、BFS の時と同様にハードウェアを扱うために必要な関数が DSL のマクロ関数によって隠蔽されている。

最後に、グラフ処理プログラムの理想的な記述である図 2.10 の簡潔かつ直感的に記述された BFS プログラムに、本 DSL がどこまで近づけたのか確認するための比較をする。この比較は図 3.3 に載せた提案 DSL により記述された 2 つの BRAM を載せたメモリコントローラを用いる BFS プログラムを用いて比較する。

図 3.3 1-9 行目

関数の引数や関数開始時に必要な記述が増えている。ただし、この部分の記述はフレームワークが必要としており、グラフアルゴリズムによって変わることはないため記述に労力を用いるというわけではない。

図 3.3 13-16 行目

BFS の最初のループ部分は並列化できないため、`while` ループ内から取り出す必要がある。図 2.10 と比較してここは冗長であると言える。

図 3.3 11,12 行目、18-24 行目

マクロによってコードブロックがどういう機能を持つのかをユーザーが指定する必要がある。また `queue` に対するアクセスをマクロに書き換える必要がある。

以上より、図 2.10 で提示した簡潔かつ直感的に記述されたプログラムよりは記述量が増えているものの、並列グラフ処理アルゴリズムで用いられるモデルによる記述や、高位合成ツールで直接変換できるように書かれた高級言語によるプログラムよりは簡潔かつ直感的に記述可能だと言える。

4.5　FPGA 上での動作検証

提案 DSL で書かれたプログラムが FPGA 上で動作することを確認する。図 4.13 で示した Bellman-Ford アルゴリズムに必要最小限の変更を加えるのみで、FPGA 評価基板上で実際に動作することを確認した。

FPGA 評価基板として PYNQ-Z1[29] ボードを用いた（図 4.15）。この評価基板には Zynq-7000 シリーズの FPGA (ZYNQ XC7Z020-1CLG400C) が搭載されている。ARM プロセッ

サと FPGA がワンパッケージに収められたデバイスであり、プロセッサでは Linux OS が搭載されている。OS からはメモリマップド I/O (MMIO) 経由で FPGA 上のハードウェアロジック (IP コア) の制御を行うことが可能である。また、ハードウェア的にはプロセッサが IP コアの制御を行うための AXI という接続インターフェースが存在する。

本検証ではメモリコントローラと Bellman-Ford プログラムから生成された回路を、Arm プロセッサに接続するために AXI インターフェースに対応させた上で FPGA に搭載した。具体的にはプロセッサから BRAM にグラフ情報を転送するために、既存の AXI-Lite インターフェースに対応した BRAM コントローラを用いた。またプロセッサが DSL によって生成された回路からアルゴリズムの解を受信するために AXI-Stream インターフェースを用いた。

PYNQ-Z1 では PYNQ フレームワークを用いることでき、Python 言語と Jupyter Notebook[30] でハードウェアロジックを制御することができる。Jupyter Notebook とはプログラム断片（セル）とテキストや図面が一体化したドキュメントを作成するための Web アプリケーションであり、対話的プログラム実行環境を提供する Python 言語と親和性が高い。IP コアの制御を行う Python ライブラリと組み合わせることで、ユーザーはより便利に IP コアの開発をすることができる。

検証について説明する。検証に用いたグラフを図 4.16 に示す。このグラフの始点（頂点 0）から終点（頂点 3）までの距離は 89（頂点 $0 \rightarrow 2 \rightarrow 1 \rightarrow 3$ と辿る）である。Jupyter Notebook での実行結果を示すスクリーンショットを図 4.17 に示す。青枠で示した部分が実行結果を途中結果も含めて送信するための stream ポートからの出力を表示したものであり、最後のデータが 89 となっている。なお、ここでは stream の終了を -1 で示している。以上により本フレームワークの DSL で書かれたプログラムが実際の FPGA 上で動作することが検証できた。

```
1  void binary_tree_bfs(Graph graph) {
2      int step = 0;
3      bool update = true;
4      while (update) {
5          update = false;
6          vector<Msg> outMsg = {};
7          // scatter phase
8          for (int vindex = 0; vindex < graph.size(); vindex++) {
9              scatter(vindex, graph[vindex], outMsg);
10         }
11         // shuffle phase
12         vector<vector<Msg>> inMsg(graph.size(), vector<Msg>());
13         for (int vindex = 0; vindex < graph.size(); vindex++) {
14             for (Msg msg : outMsg) {
15                 if (msg.dst == vindex) inMsg[vindex].push_back(msg);
16             }
17         }
18         // gather phase
19         for (int vindex = 0; vindex < graph.size(); vindex++) {
20             gather(vindex, graph[vindex], inMsg[vindex], update);
21         }
22     }
23     cout << "max: " << g_max << endl;
24 }
25
26 void scatter(int vindex, Vertex& v, vector<Msg>& outMsg) {
27     for (int dst : v.dst) {
28         outMsg.push_back({vindex, dst, (int)v.link});
29     }
30 }
31
32 void gather(int vindex, Vertex& v, vector<Msg>& inMsg, bool& update) {
33     bool link = false;
34     for (Msg msg : inMsg) {
35         if (msg.info) { link = true; break; }
36     }
37     if (link && !v.link) {
38         update = true; v.link = true;
39         if (g_max < v.value) g_max = v.value;
40     }
41 }
```

図 4.11: Scatter-Gather モデルにより記述された BFS プログラム

```
1  void binary_tree_bfs(
2      hls::stream<Node>& graph_fifo0,
3      hls::stream<Node>& graph_fifo1,
4      hls::stream<ap_uint<8>>& addr_fifo0,
5      hls::stream<ap_uint<8>>& addr_fifo1,
6      hls::stream<ap_int<32>>& out_r) {
7      Node n0, n1;
8      graph_fifo0.read_nb(n0);
9      graph_fifo1.read_nb(n1);
10
11     ap_int<32> max = 0;
12     addr_fifo0 << 0;
13     addr_fifo1 << 0;
14     graph_fifo0 >> n0;
15     graph_fifo1 >> n1;
16     if (max < n0.value) { max = n0.value; }
17     addr_fifo0 << n0.first;
18     addr_fifo1 << n0.second;
19
20     while (true) {
21         graph_fifo0 >> n0;
22         graph_fifo1 >> n1;
23         if (max < n0.value) { max = n0.value; }
24         if (max < n1.value) { max = n1.value; }
25         out_r << max;
26         if (n0.first != -1) addr_fifo0 << n0.first;
27         if (n1.first != -1) addr_fifo1 << n1.first;
28         if (n0.second != -1) addr_fifo0 << n0.second;
29         if (n1.second != -1) addr_fifo1 << n1.second;
30     }
31 }
```

図 4.12: BRAM を 2 つ内蔵するメモリコントローラを用い DSL を用いない BFS プログラム

```
void bellmanford(
    hls::stream<Vertex> &graph_fifo0,
    hls::stream<Vertex> &graph_fifo1,
    hls::stream<ap_uint<8>> &addr_fifo0,
    hls::stream<ap_uint<8>> &addr_fifo1,
    hls::stream<ap_int<16>> &out_r) {
    const ap_int<16> INF = 10000;
    ap_int<16> dist[GRAPH_SIZE];
    dist[0] = 0;
    for (ap_uint<3> i = 1; i < GRAPH_SIZE; i++) {
        dist[i] = INF;
    }

    for (ap_uint<3> i = 0; i < GRAPH_SIZE; i++) {
        ap_uint<3> j = 0;
        while (j < GRAPH_SIZE)
        {
            Vertex v0, v1;
            READ_SEQ(v, 2, j);
            CALCULATION_BLOCK(v, 2,
                UNLOOP_FOR(k, 2,
                    Edge e = v.outgoing_edge[k];
                    if (dist[e.dest_vertex_id] > dist[j] + e.dist) {
                        dist[e.dest_vertex_id] = dist[j] + e.dist;
                    }
                );
                j++;
            );
        }
        out_r << dist[GRAPH_SIZE - 1];
    }
}
```

図 4.13: BRAM を 2 つ内蔵するメモリコントローラを用い DSL を用いる Bellman-Ford 法プログラム

```
void bellmanford(
    hls::stream<Vertex> &graph_fifo0,
    hls::stream<Vertex> &graph_fifo1,
    hls::stream<ap_uint<8>> &addr_fifo0,
    hls::stream<ap_uint<8>> &addr_fifo1,
    hls::stream<ap_int<16>> &out_r) {
    const ap_int<16> INF = 10000;
    ap_int<16> dist[GRAPH_SIZE];
    dist[0] = 0;
    for (ap_uint<3> i = 1; i < GRAPH_SIZE; i++) {
        dist[i] = INF;
    }

    for (ap_uint<3> i = 0; i < GRAPH_SIZE; i++) {
        ap_uint<3> j = 0;
        while (j < GRAPH_SIZE)
        {
            Vertex v0, v1;
            addr_fifo0 << j;
            addr_fifo1 << j + 1;
            while (graph_fifo0.empty() || graph_fifo1.empty());
            graph_fifo0 >> v0;
            graph_fifo1 >> v1;
            for (ap_int<8> k = 0; k < GRAPH_DEGREE; k++) {
                Edge e0 = v0.outgoing_edge[k];
                Edge e1 = v1.outgoing_edge[k];
                if (dist[e0.dest_vertex_id] > dist[j] + e0.dist) {
                    dist[e0.dest_vertex_id] = dist[j] + e0.dist;
                }
                if (dist[e1.dest_vertex_id] > dist[j] + e1.dist) {
                    dist[e1.dest_vertex_id] = dist[j] + e1.dist;
                }
            }
            j += 2;
        }
        out_r << dist[GRAPH_SIZE - 1];
    }
}
```

図 4.14: BRAM を 2 つ内蔵するメモリコントローラを用い DSL を用いない Bellman-Ford 法プログラム

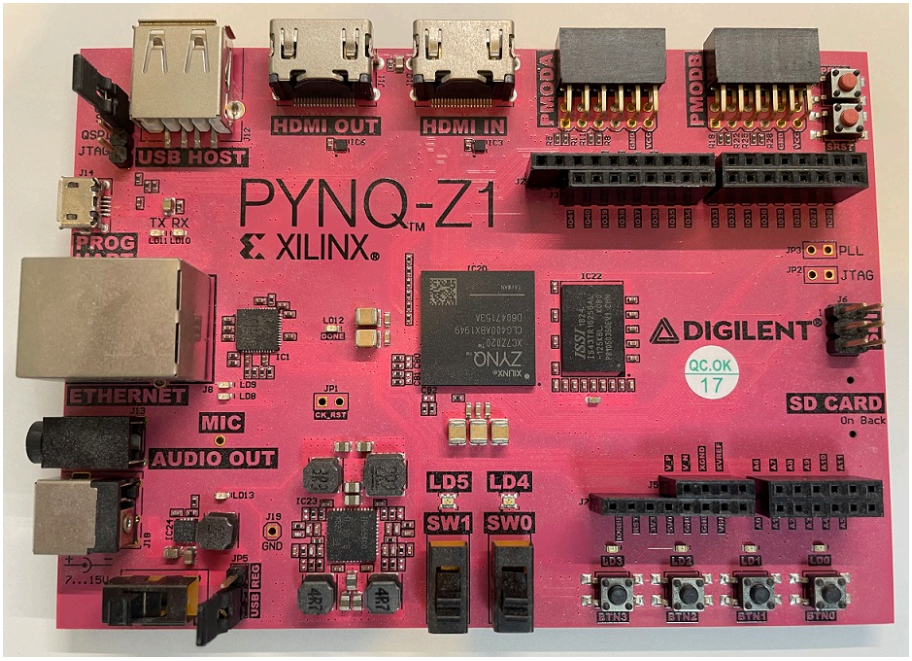


図 4.15: PYNQ-Z1

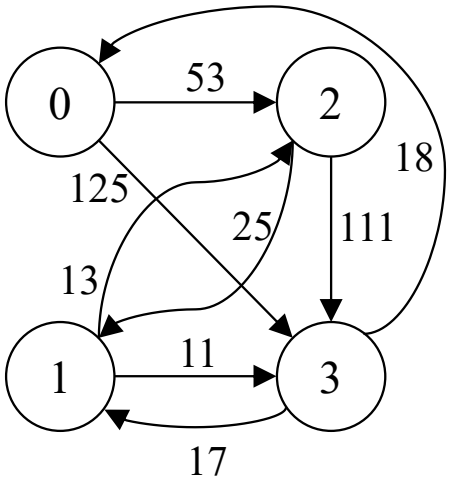


図 4.16: FPGA での動作検証に用いたグラフ

In [1]:
```
from pynq import Overlay
from pynq import MMIO
base = Overlay("./pynq_design.bit")
```

In [2]:
```
bram0 = MMIO(base_addr = 0x40000000, length = 8*1024)
bram1 = MMIO(base_addr = 0x40002000, length = 8*1024)
ip    = MMIO(base_addr = 0x40010000, length = 64*1024)
dma = base.axi_dma
```

In [3]:
```
bram0.write(0, 125 << 24 | 3 << 16 | 53 << 8 | 2)
bram1.write(0, 11 << 24 | 3 << 16 | 13 << 8 | 2)
bram0.write(4, 111 << 24 | 3 << 16 | 25 << 8 | 1)
bram1.write(4, 18 << 24 | 0 << 16 | 17 << 8 | 1)
```

In [4]:
```
import numpy as np
from pynq import allocate

buffer = allocate(shape=(50,), dtype=np.int16)
```

In [5]:
```
dma.recvchannel.transfer(buffer)
ip.write(0, 1)
dma.recvchannel.wait()
for i in range(5):
    print("Stream ID: {0}, Data: {1}".format(i, buffer[i]
buffer.freebuffer()
```

```
Stream ID: 0, Data: 125
Stream ID: 1, Data: 89
Stream ID: 2, Data: 89
Stream ID: 3, Data: 89
Stream ID: 4, Data: -1
```

図 4.17: 検証結果を示す Jupyter Notebook のスクリーンショット

第 5 章

まとめと今後の課題

5.1 まとめ

本論文では、高位合成による FPGA におけるグラフ処理ハードウェアの開発において、並列メモリアクセスやハードウェアに関する知識が必要になってしまうことを解決するために、DSL を用いた FPGA グラフ処理のための頂点への並列アクセスを可能にするフレームワークを提案した。本フレームワークは FPGA を使ったグラフ処理の記述を支援するための DSL を提供する。提案 DSL は演算の並列化を実現し、フレームワークのハードウェアに強く依存している部分を隠蔽する。この DSL は C 言語のマクロ機能を用いて実装された。また、本フレームワークは様々なグラフ処理アルゴリズムにおいて頂点データを保管し、並列メモリアクセスを実現するメモリコントローラを提供する。このメモリコントローラは主に verilog 言語を用いて実装された。以上のフレームワークによって、ユーザーが FPGA を使ったグラフ処理の高速化を手軽に実現できるようにした。本研究で行った評価により、DSL によって簡潔なグラフ処理アルゴリズムの記述を用いた FPGA グラフ処理が設計できることと、本フレームワークにより 60% 程度の処理効率向上を見込めることが分かった。

5.2 今後の課題

今後の課題の 1 つ目として、頂点データの配置戦略について述べる。現段階ではインデックスの偶奇でどちらの BRAM に配置するか決定する方法のみを実装し評価した。今回評価に用いたグラフとグラフアルゴリズムでは、インデックスの偶奇による分割配置が効率的だが、一般のグラフでもそうであるとは言えない。頂点データ配列を区切る周期を事前に決定し、その区間ごとにどちらの BRAM に配置するか決定する方法や、ランダムに配置する方法なども有効なのではないかと予想される。様々な配置戦略が考えられるとき、入力されたグラフに対してどのような配置戦略が最適なのかわかる方法があると、プログラマビリティの高いフレームワークとなる。その方法として、以下のようなプロセスで最適配置戦略を見つけられるのではないかと期待している。

1. フレームワークは様々な配置戦略を用意する。

2. グラフの一部分を FPGA に配置する前に CPU 上でグラフ処理する。
3. 一部分の頂点データに対するアクセスパターンをとる。
4. 用意した配置戦略とアクセスパターンを比較して、最も並列性が高くなる戦略を採用する。

この方法では、グラフの一部分に対して最適な配置戦略が、グラフ全体でも最適なのではないかという仮説を用いており、さまざまなグラフに対する評価を行う必要がある。

今後の課題の 2 つ目として、よりよい DSL の実現方法について述べる。よりよい DSL の実現とは図 3.3 に載せたプログラムを図 2.10 と同じかほとんど変わらないほど簡潔かつ直感的にすることである。これには 2 つの課題点がある。

- アルゴリズム中の各演算を同時に複数回行っても問題ないかどうかを静的に調べる必要がある。
- マクロを用いた DSL の設計では本研究で示したものが限界であり、並列化のためのプログラムの一部を複製する別の方法を考える必要がある。

1 つ目の課題点に近い問題として、LegUp[22] はプログラム中の演算のうち並列に実行できる部分か探す問題を、LLVM の最適化パスでエイリアス解析によって命令の独立性を調べることによって取り組んでいた。これに近い解決方法が取れるのではないかと期待する。この手法をとる場合、LLVM 最適化パスを 2 つ目の課題点の解決にも用いることができると考える。最適化パスにおいてプログラムの複製を提案 DSL のマクロより簡潔な記述方法で指示することができるからである。

発表文献と研究活動

(1) 三冨秀和, 穐山空道, 山崎徹郎, 千葉滋. FPGA グラフ処理のための頂点アクセス並列化によるプログラマビリティの高い HLS フレームワーク. システム・アーキテクチャ研究発表会, 2022.10.11.

参考文献

[1] Tiago T. Jost, Gabriel L. Nazar, and Luigi Carro. Scalable memory architecture for soft-core processors. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pp. 396–399, 2016.

[2] Miguel E. Coimbra, Alexandre P. Francisco, and Luís Veiga. An analysis of the graph processing landscape. 2019.

[3] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, Vol. 30, No. 1, pp. 107–117, 1998. Proceedings of the Seventh International World Wide Web Conference.

[4] František Duchoň, Andrej Babinec, Martin Kajan, Peter Beňo, Martin Florek, Tomáš Fico, and Ladislav Jurišica. Path planning with modified a star algorithm for a mobile robot. *Procedia Engineering*, Vol. 96, pp. 59–69, 2014. Modelling of Mechanical and Mechatronic Systems.

[5] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, Vol. 16, No. 1, p. 87–90, 1958.

[6] Robert Meusel. The Graph Structure in the Web – Analyzed on Different Aggregation Levels. *Journal of Web Science*, Vol. 1, No. 1, pp. 33–47, August 2015.

[7] Newton, Virendra Singh, and Trevor E. Carlson. Pim-graphscc: Pim-based graph processing using graph's community structures. *IEEE Computer Architecture Letters*, Vol. 19, No. 2, pp. 151–154, 2020.

[8] Hao Lu, Mahantesh Halappanavar, and Ananth Kalyanaraman. Parallel heuristics for scalable community detection. October 2014.

[9] Douglas Gregor and Andrew Lumsdaine. The parallel bgl: A generic library for distributed graph computations. *Parallel Object-Oriented Scientific Computing (POOSC)*, 01 2005.

[10] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, p. 135–146, New York, NY, USA, 2010. Association for Computing Machinery.

[11] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, Vol. 51, No. 1, p. 107–113, jan 2008.

[12] 上野晃司, 鈴村豊太郎, 松岡聡. Pregel グラフ処理系におけるメッセージ配送最適化. 情報処理学会論文誌コンピューティングシステム（ACS）, Vol. 9, No. 1, pp. 30–40, mar 2016.

[13] Srinidhi Kestur, John D. Davis, and Oliver Williams. Blas comparison on fpga, cpu and gpu. In *2010 IEEE Computer Society Annual Symposium on VLSI*, pp. 288–293, 2010.

[14] Jinhong Zhou, Shaoli Liu, Qi Guo, Xuda Zhou, Tian Zhi, Daofu Liu, Chao Wang, Xuehai Zhou, Yunji Chen, and Tianshi Chen. Tunao: A high-performance and energy-efficient reconfigurable accelerator for graph processing. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pp. 731–734, 2017.

[15] Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. Graph processing on gpus: A survey. *ACM Comput. Surv.*, Vol. 50, No. 6, jan 2018.

[16] Mohammad Bakhshalipour, Seyed Borna Ehsani, Mohamad Qadri, Dominic Guri, Maxim Likhachev, and Phillip B. Gibbons. Racod: Algorithm/hardware co-design for mobile robot path planning. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, p. 597–609, New York, NY, USA, 2022. Association for Computing Machinery.

[17] 天野英晴. FPGA の原理と構成. オーム社, 2016.

[18] ザイリンクス株式会社. Ultrascale アーキテクチャ メモリ リソース ユーザー ガイド (ug573), 2021.

[19] ザイリンクス株式会社. Spartan-7 シリーズ fpga データシート: 概要 (ds180), 2020.

[20] サラ・L・ハリス, デイビッド・ハリス. ディジタル回路設計とコンピュータアーキテクチャ [RISC-V 版]. 株式会社エスアイビー・アクセス, 2022.

[21] 高村政孝. 高位合成を用いた FPGA の開発. OKI テクニカルレビュー, 2015.

[22] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D. Brown, and Jason H. Anderson. Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems. *ACM Trans. Embed. Comput. Syst.*, Vol. 13, No. 2, sep 2013.

[23] Philippe Coussy, Daniel D. Gajski, Michael Meredith, and Andres Takach. An introduction to high-level synthesis. *IEEE Design & Test of Computers*, Vol. 26, No. 4, pp. 8–17, 2009.

[24] ザイリンクス株式会社. Vitis 統合ソフトウェア プラットフォームの資料: アプリケーション アクセラレーション開発 (ug1393), 2022.

[25] Xinyu Chen, Hongshi Tan, Yao Chen, Bingsheng He, Weng-Fai Wong, and Deming

Chen. ThunderGP: HLS-Based Graph Processing Framework on FPGAs. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '21, p. 69–80, New York, NY, USA, 2021. Association for Computing Machinery.

[26] Yuze Chi, Licheng Guo, and Jason Cong. *Accelerating SSSP for Power-Law Graphs*, p. 190–200. Association for Computing Machinery, New York, NY, USA, 2022.

[27] Charles Eric LaForest and J. Gregory Steffan. Efficient multi-ported memories for fpgas. In *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '10, p. 41–50, New York, NY, USA, 2010. Association for Computing Machinery.

[28] Vasiliki Kalavri, Vladimir Vlassov, and Seif Haridi. High-level programming abstractions for distributed graph processing. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 30, No. 2, pp. 305–324, 2018.

[29] Pynq: Python productivity. `http://www.pynq.io/`.

[30] Project jupyter. `https://jupyter.org/`.

謝辞

本研究を進めるにあたり、毎週のミーティングにてご指導いただいた本学千葉滋教授、立命館大学穐山空道准教授、本学山崎徹郎特任助教に心から感謝いたします。

千葉滋教授には研究に関すること、論文や発表に関することについて広くアドバイスをしていただきました。またメンタル面においても快く相談に乗って下さりました。他にも様々なご迷惑をお掛けしましたが、私を見放さずにここまでご指導し続けていただきありがとうございます。

穐山空道准教授には研究の進捗確認や方針、あるいは論文や発表に関して詳細にご指導いただきました。研究会での論文執筆・発表練習において、あの激励が無ければ間違いなく途中で心が折れていたと思います。プライベートな悩みや就活に関することについても相談に乗っていただきました。

山崎徹郎特任助教には私の研究に対する多角的な指摘や、研究に関する考え方についてのご指導をいただきました。研究会論文の執筆に関しても多くのご助力をいただきました。時折されていたプログラミング言語に関する研究の雑談も大変面白く聞かせていただきました。

また、本学鵜川始陽准教授を始め、本学千葉研究室の先輩方、同期の依田和樹君、白石誠君、横井駿平君、スタッフの皆様に様々なご助力をいただきました。本当にありがとうございました。

プライベートでは、大学のサークルの仲間たちや高校以来の友人たちに食事に誘っていただいたりオンラインコミュニケーションツールを用いて通話しながら遊んだりしました。コロナ渦であまり人と会えない状況では、このような助けがなければメンタルが持たず研究に支障が出ていたと思います。いつもありがとうございます。

最後に、私をこれまでずっと見守り、大学院まで様々な面で支援して下さった母と亡き父に感謝を捧げます。