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

三冨 秀和[1,a)]　穐山 空道[2]　山崎 徹郎[1]　千葉 滋[1]

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、リアルタイム性や低消費電力を求めて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（HLS）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。

1. 序論

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、その消費電力を抑えることが課題の一つであった。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、処理を効率化し、エネルギー効率を上げることができ、盛んに研究が行われている。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。このDSLは、一般のプログラマがメモリ並列化を意識せずに、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見つけ出し、それらを複数の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。

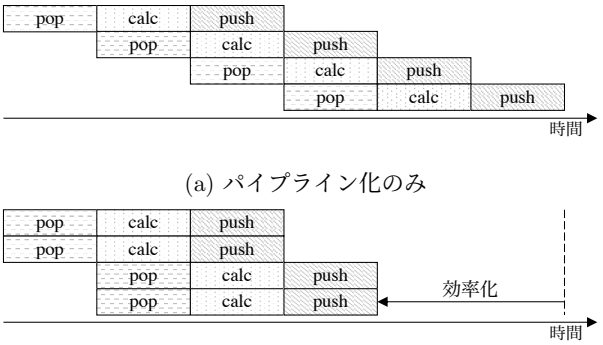
本論文での評価によって、提案フレームワークにより生成されたFPGA設計は、Vivado Simulator上でBRAMに並列アクセスしながらグラフ処理されていることが確認された。また、Scatter-Gatherのような並列化可能なアルゴリズムの記述と比較して、本フレームワークを利用すれば簡単に開発を行うことができる見込みがある。

2. ハードウェアでのグラフ処理

グラフ処理をハードウェアで行うことは重要である。グラフ処理は自律型ロボットの経路計画のような組み込み分野 [2] や、あるいはウェブ検索 [3] のような商用サービスなどで、タスクを定式化するために用いられている。従来はCPUのような汎用プロセッサ上やクラウドコンピューティングにて処理が行われることが多く、アプリケーションに特化したハードウェア上で処理を行う研究が為されてきた。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。汎用プロセッサは汎用的な問題を柔軟に解決するために設計されており、そのために特定のタスクを行う場合は過剰なエネルギーリソースを消費してしまう [4]。CPU、GPU、FPGAでBasic Linear Algebra Subroutinesのエネルギー効率を比較した研究 [5] 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。またハードウェアでのグラフ処理での応用例として、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 [6] がある。

ハードウェアでのグラフ処理の開発にはFPGAが便利

[1] 東京大学
The University of Tokyo
[2] 立命館大学
Ritsumeikan University
[a)] mitomi@csg.ci.i.u-tokyo.ac.jp



(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 1: メモリアクセスの並列化

である。なぜなら FPGA は再構成可能であるからである。ASIC などの再構成不可能な IC と比較すると、開発途中で試行錯誤しながら回路情報を書き換えることができるため、開発サイクルを短くしやすい。デメリットとして、チップ当たりのコストが高いことや、性能が低いことが挙げられる。

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 [7]。

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 1 (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。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 1 (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。

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

```
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) {
14             max = n.value;
15         }
16         if (n.first != -1) {
17             q.push(n.first);
18         }
19         if (n.second!= -1) {
20             q.push(n.second);
21         }
22     }
23     return max;
24 }
```

図 2: 簡潔かつ直感的に記述された BFS プログラム

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。

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 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。

このような理想的な記述ができる FPGA でグラフ処理を実行するための手法は我々の知る限りない。並列にグラフ処理を行うフレームワークの設計として、グラフ処理を隣接行列に対する行列計算の形に変換し入力するものや、アルゴリズムを Scatter-Gather モデルで記述させるものが一般的である。どちらの設計にせよ、ユーザーはグラフやアルゴリズムを非直感的な形に書き換える必要があり、図 2 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。これらの設計手法については 5 章で説明する。

3. 提案

3.1 概要

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 DSL によって記述されたグラフ処理を行うプログラム、各頂点の配置戦略、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。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。提案フレームワークは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 のような簡潔な記述をそのまま入力することができる。

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。頂点情報の配置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、プリプロセッサによる変換で吸収することができる。具体的な分散配置方法については 3.3 節で述べる。

図 3 に本フレームワークの概略図を示す。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。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は未完成である。そのため 4 章ではプリプロセスを手動でシミュレートすることで評価を行う。

配置戦略はグラフデータをどう分散配置するかの決め方である。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。

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。

HLS コンパイラには既存のものを利用するが、DSL プリプロセッサとメモリコントローラは我々が開発したものを使う。それぞれの詳細について本章で述べる。

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。この DSL を解釈・評価する処理系は未完成であるが、この節では今後開発する予定の DSL と 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について述べる。

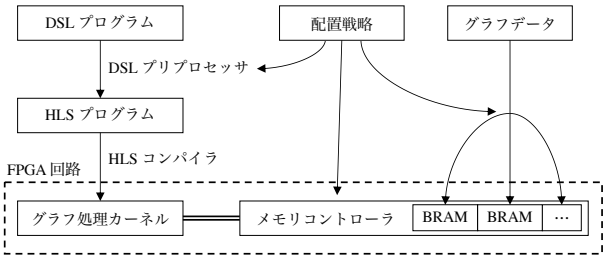


図 3: フレームワークの概要

プリプロセッサによる変換は、グラフ処理を表す DSL プログラムを入力とし、複数の頂点に並列にアクセスする HLS 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) コードブロックの二重化
(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

図 4 に図 2 の幅優先探索のプログラムの while ループの中を二重化した例を示す。図 2 では 1 つだった `max` を更新する計算や子要素の queue への挿入などが図 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。

図 5 に図 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 `data_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。`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 `data_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。

また、図 4 と 図 5 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。

```
1   while (!q.empty()) {
2       // memory access block
3       Node n1 = graph[q.front()];
4       q.pop();
5       Node n2 = graph[q.front()];
6       q.pop();
7       // calculation block
8       if (max < n1.value) {
9           max = n1.value;
10      }
11      if (max < n2.value) {
12          max = n2.value;
13      }
14      // memory access block
15      if (n1.first != -1) {
16          q.push(n1.first);
17      }
18      if (n1.second!= -1) {
19          q.push(n1.second);
20      }
21      if (n2.first != -1) {
22          q.push(n2.first);
23      }
24      if (n2.second!= -1) {
25          q.push(n2.second);
26      }
27  }
```

図 4: マクロによる変換過程

```
1   while (true) {
2       data_fifo0 >> n0;
3       data_fifo1 >> n1;
4       if (max < n0.value) {
5           max = n0.value;
6       }
7       if (max < n1.value) {
8           max = n1.value;
9       }
10      if (n0.first != -1) {
11          addr_fifo0 << n0.first;
12      }
13      if (n0.second!= -1) {
14          addr_fifo1 << n0.second;
15      }
16      if (n1.first != -1) {
17          addr_fifo0 << n1.first;
18      }
19      if (n1.second!= -1) {
20          addr_fifo1 << n1.second;
21      }
22  }
```

図 5: メモリアクセス用ライブラリによる変換過程

DSL は C++言語のマクロとライブラリを利用した Embedded DSL（EDSL）となる予定である。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 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。

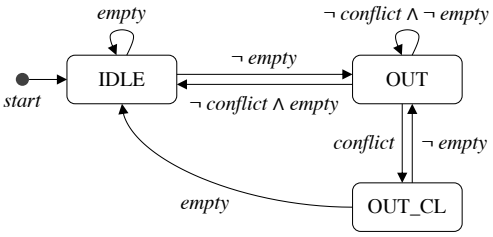


図 6: 有限状態機械の状態遷移図

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 6 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

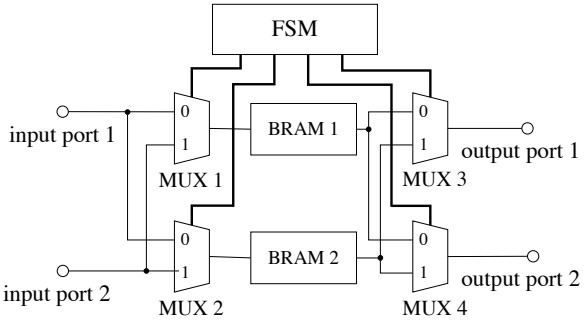


図 7: メモリコントローラ回路の概略

表 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

表 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

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 7 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 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 2 に、OUT_CL 状態でのマルチプレクサの選択信号表を表 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。

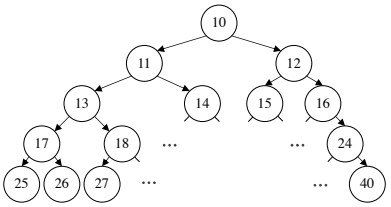
4. 評価

4.1 評価対象

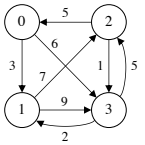
二分木に対する幅優先探索と、Bellman-Ford アルゴリズムを用いて評価を行った。評価は Vivado のシミュレータ上で行う都合、小さなグラフに対して適応することにした。

表 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



(a) 2 分木幅優先探索



(b) Bellman-Ford

図 8: 今回の評価で用いたグラフ

表 4: シミュレーター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
	Clock Period	10ns
Vivado	バージョン	2020.2
	Project Part	xc7z020clg400-1

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 8 (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 8 (b) に示す。各辺に添えた数字はその辺の重みである。

また、表 4 に今回のシミュレーションに用いた環境を示す。HLS コンパイラとして Vitis HLS を用い、HLS から回路ブロックの形で export して Vivado で読み込むことでシミュレーションを行った。

4.2 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。

この評価では、すべての頂点データを 1 つの BRAM に置いた場合と、3.3 節で述べたメモリコントローラを用いて 2 つの BRAM に置いた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。実行サイクルの計測は、Vivado のシミュレータ上で行った。なお、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 9 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。

二分木幅優先探索での計測方法を説明する。図 9 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 9 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 10 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 11 に計測開始から計測終了までにかかったサイクル数を示す。縦軸はサイクル数であり、横軸は計測条件を示している。提案したメモリコントローラを用いた場合、用いなかったときに比べて、2 分木幅優先探索で約 68%、Bellman-Ford で約 64%のサイクル数で処理することができた。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。

4.3 DSL の簡単さの評価

提案フレームワークの入力となるグラフアルゴリズム記述と、Scatter-Gather モデルによる記述を比較する。Scatter-Gather モデルによるグラフアルゴリズムの記述について Kalavri ら [8] の論文の Algorithm4 に記載がある。これと今回の簡潔な BFS プログラム（図 2）を比較して簡単になっているということを説明する。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 の形に変形する。
- scatter 関数と gather 関数をそれぞれ頂点間通信としてインターフェースを利用して記述する。

これに比較して今回の BFS プログラムは特に変形などせずにフレームワークで利用可能である。

5. 関連研究

5.1 ThunderGP[9]

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 関数をそれぞれ書き、グラフデータと合わせてこのシステムの入力となる。各関数に対応する演算回路は FPGA 上にそれぞれ複数配置される。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キャッシュされる。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。一方我々の提案フレームワークは、プログラマがアルゴリズムを直感的に記述できるように設計されている。

5.2 SPLAG[10]

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。

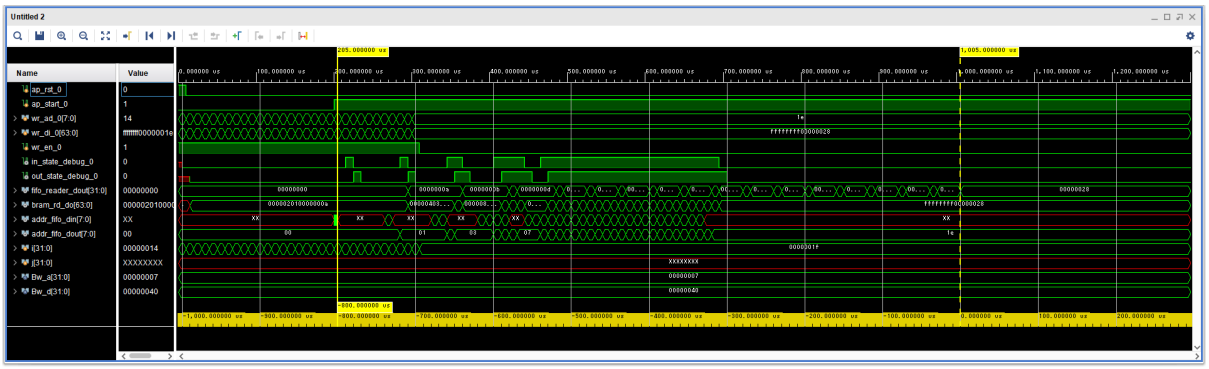


図 9: 幅優先探索を BRAM1 つでシミュレーションを行った時に出力される waveform

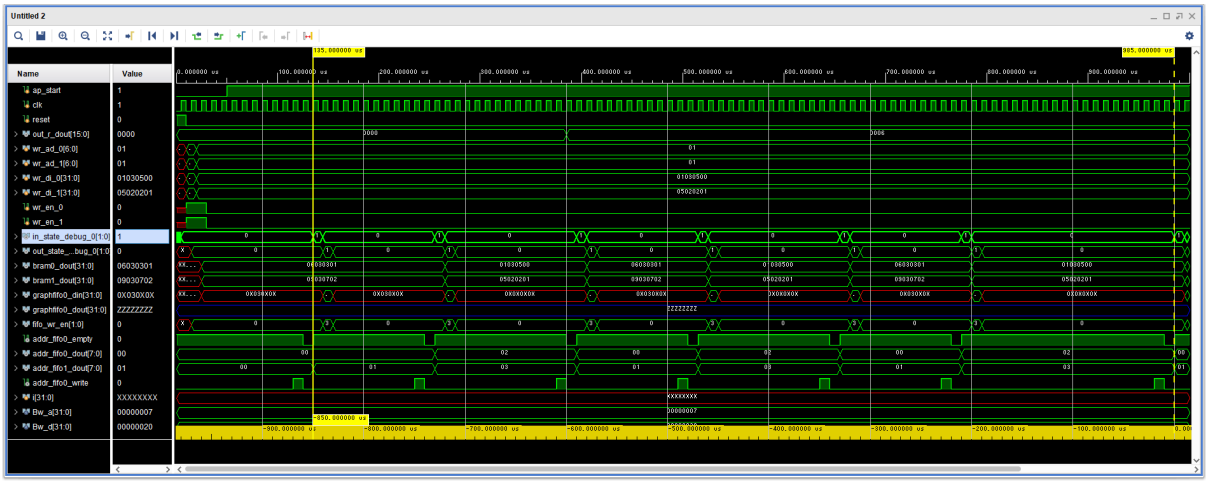


図 10: Bellman-Ford を BRAM2 つでシミュレーションを行った時に出力される waveform

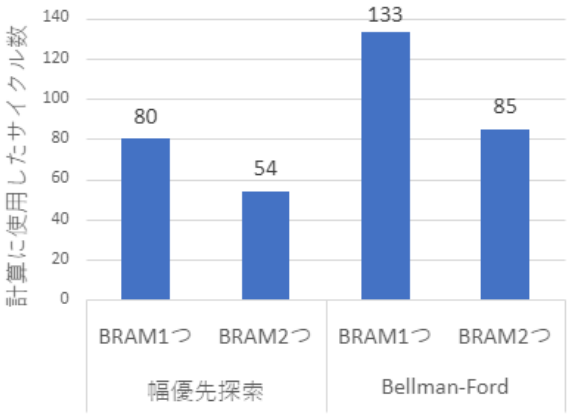


図 11: 並列性の評価結果

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。一方我々の手法では、DSL によって様々なアルゴリズムを表現することが可能である。

6. 結論と今後の課題

本論文では、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。本フレームワークには DLS によるアルゴリズムの記述と並列アクセスを実現するメモリコントローラの 2 つの要素が含まれている。メモリコントローラは実装されており、実際に 60~70%の処理効率向上を見込める。それに対して DSL は未だ実装されておらず、今後 3.2 節で述べたような DSL を実装する予定である。その際、グラフ処理の中心となるループ内部のコードが複製されることとなるが、そのループをマクロで指定するような形で記述できるような方法で考えている。さらに、3.2 節では queue の操作がアルゴリズムの中心となる幅優先探索について考えたが、一般の場合はその操作限りではない。例えばすべての頂点を順番にたどっていく場合や、スタックに対する操作などが考えられるが、現状では先ほどのマクロでそれを指定し、複製の方法を変更することを考えている。また、グラフ頂点アクセスをフレームワークによって用意されたインターフェースを用いて、FIFO に対するアクセスに変換することを考えている。

次に、頂点データの配置戦略について今後の課題を述べる。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。

参考文献

[1] Jost, T. T., Nazar, G. L. and Carro, L.: Scalable memory architecture for soft-core processors, *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pp. 396–399 (online), DOI: 10.1109/ICCD.2016.7753312 (2016).

[2] Bakhshalipour, M., Ehsani, S. B., Qadri, M., Guri, D., Likhachev, M. and Gibbons, P. B.: RACOD: Algorithm/Hardware Co-Design for Mobile Robot Path Planning, *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, New York, NY, USA, Association for Computing Machinery, p. 597–609 (online), DOI: 10.1145/3470496.3527383 (2022).

[3] Zhou, S., Chelmis, C. and Prasanna, V. K.: Optimizing memory performance for FPGA implementation of pagerank, *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pp. 1–6 (online), DOI: 10.1109/ReConFig.2015.7393332 (2015).

[4] 吉澤 慶，石原 亨，小野寺秀俊：組み込みアプリケーションにおける汎用プロセッサと専用ハードウェアの性能解析—消費エネルギーと処理速度および回路規模の定量的評価—，DA シンポジウム 2016 論文集，Vol. 2016, No. 19, pp. 103–108 (2016).

[5] Kestur, S., Davis, J. D. and Williams, O.: BLAS Comparison on FPGA, CPU and GPU, *2010 IEEE Computer Society Annual Symposium on VLSI*, pp. 288–293 (online), DOI: 10.1109/ISVLSI.2010.84 (2010).

[6] Zhou, J., Liu, S., Guo, Q., Zhou, X., Zhi, T., Liu, D., Wang, C., Zhou, X., Chen, Y. and Chen, T.: TuNao: A High-Performance and Energy-Efficient Reconfigurable Accelerator for Graph Processing, *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pp. 731–734 (online), DOI: 10.1109/CCGRID.2017.114 (2017).

[7] Harris, S. L. and Harris, D.: *Digital Design and Computer Architecture, RISC-V Edition*, Morgan Kaufmann Publishers (2021).

[8] Kalavri, V., Vlassov, V. and Haridi, S.: High-Level Programming Abstractions for Distributed Graph Processing, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 30, No. 2, pp. 305–324 (online), DOI: 10.1109/TKDE.2017.2762294 (2018).

[9] Chen, X., Tan, H., Chen, Y., He, B., Wong, W.-F. and Chen, D.: ThunderGP: HLS-Based Graph Processing Framework on FPGAs, *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '21, New York, NY, USA, Association for Computing Machinery, p. 69–80 (online), DOI: 10.1145/3431920.3439290 (2021).

[10] Chi, Y., Guo, L. and Cong, J.: *Accelerating SSSP for Power-Law Graphs*, p. 190–200 (online), available from ⟨https://doi.org/10.1145/3490422.3502358⟩, Association for Computing Machinery (2022).