技術・研究報告

高位合成による FPGA アクセラレータの試用と考察

若谷彰良

甲南大学 知能情報学部
神戸市東灘区岡本８－９－１, 658-8501

(受理日 2022 年 11 月 28 日)

概 要

近年, 高速計算のためのアクセラレータとして FPGA が注目されている. 一般に, FPGA の再構成にはハードウェア記述言語 (HDL: Hardware Description Language) が用いられ, ハードウェア技術者以外には難しいものとされているが, 昨今, C 言語や C++言語を利用した高位合成が普及しつつあり, そのハードルは下がってきている. しかし, そのプログラミング環境の使いやすさや性能向上によるアドバンテージについては未知数な部分も多い. 本稿では, その試用を通して実用性及び可用性の予備評価と考察を行なう.

キーワード: 高速計算, 行列計算, アンロール, 並列化, 論理合成

1 はじめに

ムーア則の限界にともない, 単体でのプロセッサの性能向上が鈍化している. 量子ゲート方式や量子アニーリング方式による量子計算が, 次世代の計算方式として注目されているが, 実用レベルで実現および普及するにはまだまだ乗り越えるべき障壁は多い [1]. そこで, 既存のノイマン方式によるコンピュータの改良による性能向上が図られ, 単体プロセッサでは複数のプロセシングコアによるマルチコアプロセッサや, 演算器中心のコアを大量に搭載したグラフィックプロセッサ (GPU: Graphics Processing Unit) を利用した高速計算などが主流となっている. しかし, GPU は消費電力が大きく, 消費電力と高性能計算の両立は解決すべき問題の一つである.

一方, FPGA (Field-Programmable Gate Array) は, 消費電力が低く, ユーザ自身が論理回路を再構成できる LSI で, VHDL や Verilog HDL などのハードウェア記述言語 (HDL: Hardware Description Language) を用いて, LSI の構成を記述することができるものである. 主要メーカとして, Intel[1] や AMD[2]がある. さらに, FPGA 自体の大規模化とともに, 高位合成 (HLS: High-Level Synthesis) 技術により, ハードウェア記述言語による設計だけでなく, 一般的なプログラミング言語による利用も可能となっており [2], HPC (High Performance Computuing) の分野 [3] やグラフ計算の分野 [4] など, FPGA の利用範囲は拡大している. FPGA の高位合成の環境として, Intel は oneAPI [5] を, AMD は Vitis [6] を提供している.

[1]Altera は 2015 年に Intel 傘下になった.
[2]Xilinx は, 2022 年に AMD の小会社となっている.

また, ACRi (アダプティブコンピューティング研究推進体, Adaptive Computing Research Initiative) では, Web 上での FPGA の利用機会の提供を行っており, 利用の裾野は着実に拡がっている [7].

本稿では, AMD の FPGA Alveo U250 を対象に, Vitis での C 言語と OpenCL による高位合成を用いて, FPGA をアクセラレータとしての利用の可用性について, いくつかの行列計算をもとに予備評価し, そのメリット及びデメリットについて考察する, 第 2 章では, AMD での高位合成の環境である Vitis による高位合成手法と典型的なプログラミング方法について述べ, 第 3 章では, 行列の和及び積に関して, FPGA (Alveo U250 [8]) における性能評価を通して, FPGA アクセラレータの可用性について考察し, 第 4 章でまとめる.

2 Vitis による高速化

AMD は FPGA の高位合成のために Vitis を提供している. Vitis による FPGA 計算においては, CPU (以下, ホストと呼ぶ) で実行する部分 (以下, ホストプログラムと呼ぶ) と, ホストプログラムから起動された FPGA (以下, デバイスと呼ぶ) 上での実行内容を記述する部分 (以下, カーネルプログラムと呼ぶ) からなる. ホストプログラムでは, FPGA に対する各種命令を送出するためのコマンドキューや, CPU と FPGA でデータ共有するメモリ空間の設定やデータ移動などを, OpenCL [9] を用いて表す. OpenCL は, Khronos グループが制定した標準規格で, FPGA だけでなく, GPU やマルチプロセッサなど, ホモジニアス及びヘテロジニアスな並列システムの動作を記述することができる.

FPGA の基本的な構造は, 任意の論理演算を実現する LUT (Look-Up Table) とメモリの役割の FF (Flip-Flop) からなる論理ブロック (BLE: Block Logic Element) が大量にあり, それらをスイッチブロックやコネクションブロックで接続することで任意の LSI を実現する構成になっている. また, Aleveo U250 のような最新の FPGA では, メモリ専用の BRAM (Block RAM) や DSP が別途搭載され, 高機能な論理回路を実現できるようにしている. FPGA ボードには, 大容量の DDR メモリが搭載されており, BLE からアクセス可能であるが, 大量の論理回路を並列動作させるために, 複数の論理回路を構成するとともに, 高帯域のメモリアクセスをするために BRAM を有効活用することが重要となる. ただし, BRAM はホストから直接アクセスできないので, DDR メモリを介して間接的に読み書きすることが必要である.

一般的な FPGA の設計手順は, 動作レベルの記述を高位合成により RTL (Register Transfer Level) 記述に変換し, RTL 記述を論理合成してネットリストを作成される. RTL 記述を手動で書く場合はハードウェア記述言語を用いる. ネットリストは LUT や BRAM 等の FPGA の構成要素に割り当てられ, その後, 配置配線されてコンフィギュレーションファイルに変換される. コンフィギュレーションファイルはアプリケーション開始時にハードウェア上にロードされ, ホストプログラムからカーネルプログラムが呼び出される形式でプログラム実行される.

ホストとデバイスの初期設定は以下の通りである. まず, cl::Platform::get でプラットフォーム全体を取得し, Xilinx のデバイス名を持つプラットフォームを見つける. cl::Context でコンテキストを作成し, cl::CommandQueue で作成したコンテキストとデバイスを結び付けたコマンドキューを宣言し, コンフィギュレーションファイル (デフォルトでは, binary_container_1.xclbin) をロードする. cl::Program で, デバイスとコンテキストとロードしたプログラムを設定する. cl::Kernel で, 設定したプログラムと起動するカーネルプログラムの関数（以下, カーネル関数と呼ぶ）の名前を結合し, このカーネル関数

がホストプログラムから呼ばれた時に, コンフィギュレーションファイルを用いてデバイスの再構成とカーネル関数の起動を行うように設定する.

ホストプログラムからカーネル関数を起動をする手順は以下の通りである. ホストとデバイスでデータの送受信をするための領域を cl::Buffer で確保し, その領域をホスト側から読み書きするポインタをコマンドキューの enqueueMapBuffer メソッドでマップする. 実際のホストとデバイス間のデータ移動は enqueueMigrateMemObjects で行われる. カーネルの起動の前に, setArg メソッドでカーネル関数の引数を設定し, enqueueTask メソッドでカーネル関数を呼び出す. さらに, コマンドキューの finish メソッドで, それ以前のコマンドが完了したことを確認する.

なお, Vitis におけるビルドには 3 種類あり, Emulation-SW は OpenCL プログラムをホストで実行するもので FPGA は全く使わない. Emukation-HW は OpenCL プログラムをネットリストにまで変換し, ホスト上のシミュレータで実行するものであり, これも FPGA での実行は行わない. Hardware は FPGA で実行できるコンフィギュレーションファイルを作成し, FPGA を使って実行する. 後述するように, Emulation-SW および Emulation-HW は数分以内でビルドが完了するが, Hardware は 1 時間以上のビルド時間を要するので, プログラム動作の確認やハードウェア量の見積りは, Emulation-SW や Emulation-HW を有効活用する必要があり, Hardware でのビルドは最終段階のみにするように努力する.

3 行列計算の FPGA 化

3.1 高速化のための実装

FPGA による計算の高速化は, 高速化したい部分を関数として記述し, それをカーネル関数として実装することで実現できる. プログラムのカーネル関数化は前述の OpenCL を用いることになるが, それだけでは FPGA をデバイスとして使うだけで必ずしも高速化にならない. 高速化には時間並列つまりパイプライン化および空間並列, 並列化の両方が必要となる.

図 1 のプログラムを元に, 高速化の手法を説明する.

```
1 for(i=0;i<100;i++){
2  x[i]=y[i]+z[i];
3 }
```

図 1: サンプルプログラム

図 1 のプログラムは 100 個の要素を持つ配列 x, y, z に対し, y と z の加算を x に代入するものである. ここで, 配列はあらかじめ, BRAM に格納されているとする. BRAM は FPGA ボードの DDR メモリよりも高速であるが, ホストからはアクセスできない領域になるので, カーネル関数の中で, あらかじめ, DDR メモリから BRAM にコピーする動作を記述する必要がある.

このプログラムをそのまま FPGA で高位合成及び論理合成すると, x[i]=y[i]+z[i] を 100 回繰り返す回路が構成され, ホストに比べ, デバイスの方が動作周波数が低いことにより, 高速計算は実現

できない可能性が高い. 一般にホストの CPU の動作周波数は数 GHz であるが, FPGA は数百 MHz であり, 今回のシステムでは約 300MHz の回路になっている.

```
1 for(i=0;i<100;i++){
2 #pragma HLS PIPELINE
3  x[i]=y[i]+z[i];
4 }
```

図 2: パイプライン化したサンプルプログラム

そこで, for ループをそのまま繰り返すのではなく, 一定のインターバルを開けて, 連続して計算を投入することで, 高速化が図れる. これを時間並列もしくはパイプラインと呼ぶ.

それを実現するのが, 図 2 に示す, HLS プラグマ (Pragma) である. プラグマはコンパイラに何らかの指示を与える構文で, HLS, つまり高位合成 (HLS: High-Level Syhthesis) のプラグマであることを示し, PIPELINE で, 直近の for ループをパイプライン実行で実現することを指定している. for ループ 1 回のレイテンシが α とすると, 非パイプラインのプログラムのループ全体のレイテンシは 100α になるが, ループ間のインターバルが β のパイプライン化したプログラムでは, ループ全体のレイテンシは $\alpha+99\beta$ となり, $\alpha >> \beta$ の条件では, 約 α/β 倍の高速化になる. ただし, パイプライン実行を行うには, ループ要素間でデータ依存が無い条件を満たす必要があり, 例えば, `x[i]=x[i-1]+y[i]` のような計算の場合はパイプライン化できない. なお, 一般的に, $\alpha >> \beta$ は成り立つ場合が多い.

さらに, `x[i]=y[i]+z[i]` を行う論理回路を複数個実装し, 連続するループ要素を同時に実行することでも高速化が実現できる.

```
1 for(i=0;i<100;i++){
2 #define HLS UNROLL factor=4
3  x[i]=y[i]+z[i];
4 }
```

図 3: 並列化したサンプルプログラム

図 3 では, ループ要素を 4 個まとめて行うアンロール, つまり, ループを部分展開したプログラムを示している. ここでは, UNROLL プラグマを用い, factor を 4 とすることで, 4 個のループ要素を同時に実行するようにしている. つまり,`i`=0,1,2,3 のループ演算を同時に行えるように 4 個の論理回路を実装し, 次の繰り返しでは, `i`=4,5,6,7 のループ演算を行う. よって, 100 回の繰り返しの代わりに 25 回の繰り返しで完了できるので, 理想的には 4 倍の高速化となる. なお, パイプラインとは異なり, 同じ論理回路を 4 個必要とするので, 高速化の代わりにハードウェアの増加がトレードオフとなる.

しかし, アンロールによる高速化を実現するにはメモリからのデータ供給も 4 倍必要となるが, メモリのポート数の制限から, 必ずしも 4 倍のデータ供給が実現されるわけではない. そこで, メモリを異なるメモリ要素に分割し, ポート数の増加, つまり, メモリバンド幅の増加を行う必要がある.

図 4 に BRAM を分散させた並列化プログラムを示す. BRAM の分散には, ARRAY_PARTIRTION プラグマを用いる. 前述の通り, `i`=0,1,2,3 のループ演算を同時に行なうには, `x[0], x[1], x[2], x[3]` は異なる BRAM に配置されていなければならない. よって, `x[0], x[4], x[8]`. が同じ

BRAM, x[1], x[5], x[9]. が同じ BRAM のように, サイクリックに配置されるように指定することになる. 配列 y および z も同様であり, 図 4 のようなプラグマを用いて, メモリ配置を指定する. よって, x[0], x[1], x[2], x[3] が同時にアクセスできるので, 4 個のループ演算が同時に実行可能となり, 高速化される可能性が高くなる.

```
1 #define HLS ARRAY_PARTITION variable=x type=cyclic factor=4
2 #define HLS ARRAY_PARTITION variable=y type=cyclic factor=4
3 #define HLS ARRAY_PARTITION variable=z type=cyclic factor=4
4 for(i=0;i<100;i++){
5 #define HLS UNROLL factor=4
6  x[i]=y[i]+z[i];
7 }
```

図 4: 分散した BRAM を用いた並列化したサンプルプログラム

以下の節では, 行列の和および積を例にとり, ホストとデバイスの性能について予備評価する. なお, FPGA の実行においては, 実行前に, FPGA の回路の再構成に一定の時間がかかる. 実測においては, 約 3 秒の時間がかかり, 同じアプリケーションでも, 1 回目の実行と, 2 回目以降の実行で, 前者が約 3 秒長くかかる. 以下の性能評価においては, FPGA の再構成の時間を省いた, 2 回目以降の実行時間で評価を行う.

3.2 行列計算における予備評価

HPC プログラムの基本要素である行列計算を用いて, 高位合成による FPGA の高速化について予備評価を行う.

評価に用いる AMD の Alveo U250 は FPGA を搭載した PCIe 対応のアクセラレータカードであり, 16 nm プロセスによる UltraScale+ Virtex FPGA と 64 GB の DDR4 SDRAM で構成されている. FPGA には, 1728K 個の LUT (Look-Up Table), 12288 個の DSP 等が搭載されており, AMD の FPGA のラインナップの中では最上位機種の一つである. また, 比較対象とするホストは, CPU が Intel Core i9-10920X CPU @ 3.50GHz で, メモリが 64 GB であり, OS は Ubuntu 20.04.5 LTS である. Vitis のバージョンは Vitis v2022.1 (64-bit) である.

表 1 に, 100×100 の行列の加算のおける, アンロールの回数とアクセラレータによるスピードアップの結果を示す. スピードアップは, ホストのみでの実行時間を Emulation-SW でビルドした場合の時間とし, それを Hardware でビルドした場合の実行時間で除算した値で示す. 加算は, 計測の精度をあげるために 200,000 回繰り返している.

表 1: 行列和計算における高速化

UNROLL 回数	8	16	32
Speedup	3.61	6.02	11.04

アンロールの回数を 2 倍する毎に, 同時に実行できる加算が 2 倍になっている. BRAM をアンロールの個数にしたがって分散 (ARRAY_PARTITION) しており, 約 2 倍の速度向上になっている, よって, 行列加算のように, データの重なりがなく, さらにデータ依存のない単純な計算であれば, アンロールによる並列化による高速化が達成できており, 問題サイズを大きくすることで, ハードウェアの制限まで並列化できると考えられる.

次に行列積について考える. 行列積に関しては, 図 5 に示すように, i ループ, j ループ, k ループの 3 箇所に UNROLL プラグマを挿入でき, 実行時間を比較することが必要になる.

k ループは変数 val がループ依存あるので, 展開した演算毎に val の値を生成し, それを最後に合計するというようにアンロールを実現する必要であるが, Vitis が自動的に対応している. また, メモリアクセスの同時実行性を考慮するために, 配列 kernel_x は第 2 の次元, つまり, インデックス i を, 配列 kernel_y は第 1 の次元, つまり, インデックス j を分散するように BRAM 上に分散配置するプラグマを入れる.

```
1  LOOP11:for (int i=0; i<N; i++) {
2    #pragma HLS UNROLL factor=2
3    LOOP12:for (int j=0; j<N; j++) {
4      #pragma HLS UNROLL factor=2
5      int val=0;
6    LOOP13:for(int k=0; k<N; k++){
7      #pragma HLS UNROLL factor~=2
8        val=val+kernel_y[i][k]*kernel_z[k][j];
9      }
10     kernel_x[i][j]=val;
11   }
12 }
```

図 5: 行列積プログラム

表 2 に UNROLL プラグマの挿入位置によって, 行列積プログラムの実行時間がどのように変化するかを示す. 実行時間は, 100×100 の行列に対し, LOOP11 を 200 回反復する時間を計測したものとする. 図に示すように, k のループにプラグマを入れる場合の実行時間は相対的に短く, さらに j のループも同時にアンロールするケースが最速になった. よって, 以下では, プラグマ挿入位置は j と k のループとする.

表 2: 行列積計算における UNROLL プラグマ挿入場所の比較 (秒)

	i 有り		i 無し	
	j 有り	j 無し	j 有り	j 無し
k 有り	0.59	0.55	0.54	0.55
k 無し	0.89	0.95	0.91	—

表 3 に, 表 2 と同様に, 100×100 の行列の乗算のおける, j と k ループのアンロールの回数とアクセ

ラレータによるスピードアップの結果を示す. つまり, 行列積プログラムにおいて, プラグマにおける factor の値を変える. 行列和の場合と同様に, ホストのみでの実行時間を Emulation-SW でビルドした場合の時間とし, それを Hardware でビルドした場合の実行時間で除算したスピードアップを示す. 乗算は, 計測の精度をあげるために 2,000 回繰り返している.

表 3: 行列積計算における高速化

UNROLL 回数	2	4	8
Speedup	1.59	2.75	4.05

表 1 のケースと同様に, アンロール回数を 2 倍にする毎に, 約 1.5 倍以上の性能向上が得られている. 再内側ループにループ依存変数 (`val`) があり, また, データ参照の重なりがある等のオーバヘッドがあることから, 行列和より性能向上は下がっているが, 一定の並列効果はある.

3.3 考察

性能向上に関して, 表 1 及び表 3 に示すように, ホストコンピュータの実行よりも高速化が達成され, FPGA によるアクセラレータの効果が示された. しかし, ビルドにおいていくつかの問題点が生じた. まず, 行列積に関して, 表 3 のように, アンロール回数を 8 まで計測したので, さらに UNROLL 回数を増やすことでさらに性能向上する可能性があるが, 現時点の Vitis では論理合成以降のビルドが正しく行えなかった. 具体的には, 論理合成はできているが, 配置配線ができずにビルドが失敗した. さらに, アンロールしていないループにパイプラインを適用しようとしたが, ビルドが行えなかった. また, `ARRAY_PARTITION` による BRAM の分散も, factor のサイズにより, BRAM ではなく, FF に配列が配置される場合があった. ソフトウェアにおけるコンパイラとは異なる論理合成および高位合成独特な特徴があるので, 様々なケースを試すことで, 配列分散の方法の手法等のビルドの手法を確立する必要がある.

また, ビルドに要する時間に関しては, ソフトウェア技術者の感覚からは極めて長く感じられた. HPC などのアクセラレータとして GPU の活用が進んでいるが, GPU でのビルドは CPU でのビルドと大差なく, FPGA での Emulation-SW や Emulation-HW でのビルドでも数分以内であり, ソフトウェア感覚でプログラミングできる. しかし, FPGA における Hardware でのビルドは, 少なくても 1 時間, 長い場合は 2 時間近くのビルドが必要であり, 大規模なプログラムをチューニングしながらプログラミングすることは, 実用上大きな課題である.

以上のようなソフトウェア開発にはない, ハードウェア開発に特化した問題は散見されるが, 低消費電力で高速化が可能であり, 高位合成を用いれば通常のプログラミング言語でプログラミングできることは魅力的であるので, 引き続き, 実用的アプリケーションプログラムの FPGA アクセラレーションに取り組んでいきたい.

4 おわりに

次世代の計算方式として FPGA による大規模計算のアクセラレーションが注目されており, 高位合成による普通のプログラミング言語による FPGA 利用が普及しつつある. 本稿では, HPC で頻繁に用いられる行列計算の実装および実行時間の計測を通して, 実用性及び可用性の予備評価と考察を行なった.

UNROLL プラグマによる FPGA 上の論理回路の並列化により, CPU 実行に対して高速化が達成できたが, Vitis によるプログラミング環境についてはいくつかの課題も見えてきた. プログラミングのノウハウを蓄積し, 今後, 実用的なプログラムにおける性能向上を目指して FPGA 利用を進めることが課題である.

謝辞

本研究の一部は JSPS 科学研究費 (基盤研究 (C) 18K02920 (2018-2022), 基盤研究 (C) 19K03018 (2019-2022)) 及び私立大学等経常費補助金特別補助「大学間連携等による共同研究」による.

参考文献

[1] 若谷彰良, “組合せ最適化問題に対する量子アニーリング方式の試用と考察,” 甲南大学紀要 知能情報学編, vol. 15, no. 1, pp. 37-44, 2022.

[2] 長瀬雅之他, 高位合成よる FPGA 回路設計. 森北出版, 2022.

[3] 柿根尚喜, 窪田昌史, 弘中哲夫, “FPGA 向け 4 倍精度浮動小数点演算器の設計と共役勾配法による評価,” 研究報告システムと LSI の設計技術（SLDM）, 2022-SLDM-197(27), pp. 1-5. 2022.

[4] 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,” 研究報告システム・アーキテクチャ（ARC）, 2022-ARC-250(2), pp. 1-8, 2022.

[5] “Intel FPGA Add-on for oneAPI Base Toolkit,” https://www.intel.com/content/www/us/en/developer/tools/oneapi/fpga.htm (As of 2022/10/27).

[6] “Vitis 高位合成ユーザー ガイド (UG1399),” https://docs.xilinx.com/r/ja-JP/ug1399-vitis-hls (As of 2022/10/27).

[7] “アダプティブコンピューティング研究推進体,” https://www.acri.c.titech.ac.jp/wp/ (As of 2022/10/27).

[8] “Alveo U250 Data Center Accelerator Card,” https://www.xilinx.com/products/boards-and-kits/alveo/u250.html (As of 2022/10/27).

[9] “OpenCL,” https://www.khronos.org/opencl/ (As of 2022/10/27).