令和3年度 修了論文

FPGAを用いた重力多体問題の高位合成に関する最適パラメータ推定とツール間性能比較

電気通信大学 情報理工学研究科
情報・ネットワーク工学専攻 コンピュータサイエンスプログラム

学籍番号　2031140
氏名　村松 耀生
指導教員　成見 哲
指導教員　吉永 努

令和4年1月28日

概 要

複数の計算コアを有する GPU を用いた数値計算アクセラレーションが脚光を浴びているが，高周波数で大量のプロセッサを駆動することになるため，消費電力の増大が懸念されている．それに対して，再構成可能な論理デバイスである FPGA が低消費電力性と計算高速性の両立が可能 という理由から近年，データベース検索やクラウドサービスの分野で注目を集めている．

ただし，FPGA を含む大規模集積回路設計開発は，ハードウェア記述言語を利用してレジスタ転送レベルで行う必要があり，回路仕様の設計や実装，デバッグや最適化など様々な開発に大きなコストを要していた．

そこで FPGA の設計において開発期間短縮のために高位合成が使われ始めており，ハードウェアで行う処理部分を RTL ではなく C 言語等で記述出来る．更にアプリケーション全体に対し高位合成することでソフトウェアとハードウェアのインターフェイス部分も自動で合成可能である．Xilinx 社の高位合成用 FPGA 設計ツールは SDSoC から Vitis に移行したが機能が追加，削除されるなど最適化手法が変化している．

そこで本研究では，まず重力多体問題を例として同じ C のコードを元に SDSoC と Vitis で機能や生成ハードウェアの性能の違いを比較した．Vitis は自動でパイプライン化したり通信ハードウェアが高速であったが，計算速度は SDSoC に劣っていた．また，調査を進めていくと計算全体の並列化には OpenCL での実装が必要だとわかった．

次に OpenCL でコードを書き換えた後に，どの程度並列化して加速できるか調査した．低並列時の回路資源量から最大性能を出すためのパラメータ予測を行った．その結果，回路資源などの制約のない状況ではループ展開による並列処理をすることが最適であるが，計算量が少なく通信時間の全体に占める割合が大きい場合は通信部分も含めた計算ユニットを複数実装する方法が最適になる場合もあることが分かった．

目 次

1　はじめに

本章では，まず本研究を行うにあたった研究背景を述べ，次に本研究での目的を明らかにする．最後に本論文の構成を述べる．

1.1　研究背景

複数の計算コアを有する GPU(Graphics Processing Unit) を用いた数値計算アクセラレーションが脚光を浴びているが，高周波数で大量のプロセッサを駆動することになるため，消費電力の増大が懸念されている．それに対して，再構成可能な論理デバイス である FPGA(Field Programmable Gate Array) が低消費電力性と計算高速性の両立が可能という理由から近年，データベース検索やクラウドサービスの分野で注目を集めている．FPGA では任意の回路を構成できることから，従来のノイマン型計算機の枠組みを超越し，自在に計算機構を構成できる点が最大の魅力である．

ただし，FPGA を含む大規模集積回路 (Large Scale Integrated circuits:LSI) 設計開発は，ハードウェア記述言語 (Hardware Description Language:HDL) を利用してレジスタ転送レベル (Register Transfer Level:RTL) で行う必要があり，回路仕様の設計や実装，デバッグや最適化など様々な開発に大きなコストを要していた．そこで，FPGA(Field Programmable Gate Array) の大規模化に伴い，RTL レベルではなく C 言語等の高級言語で設計する高位合成が使われ始めている．近年の FPGA にはハードコアの CPU が内蔵していることも多く，また仮に内蔵していなくてもソフトコアの CPU を搭載するなどして Linux 等の高機能な OS を走らせた上で，処理の重い部分だけ専用ハードウェアで加速する場合も多い．その際 C 言語等のアプリケーションプログラムから専用ハードウェアを呼び出す部分のドライバや通信回路の設計が必要になる．Xilinx 社は 2015 年に SDSoC[1] を発表し，2019 年にはその後継となる Vitis[2] を発表した．

しかし，高位合成による振る舞いの記述は RTL より抽象度が高いため,RTL の記述に比較して記述量が少なく設計柔軟性・生産性共に優れる反面，演算に対応する演算器，変数に対応するレジスタなどを筆頭として，様々な論理資源への割り振りは高位合成ツールに委ねられる [3]．現在では高位合成によって生成される回路の面積や性能を最適化する方法が提案され，実用的なレベルの高位合成が可能になったとされている．高位合成によって回路設計開発期間の短縮と設計抽象性の更なる向上が実現されたものの，レジスタ転送レベルでの設計と比較して高位合成では算術回路の解析的な最適化を行いにくいという問題が依然として残っている．特に，数値アクセレータの開発においては，特定のスペックを満たす速度を実現することよりはハードウェアをどこまで最適化できるかと

いう点に主眼が置かれる．故に．高位合成からより性能の良い回路を如何に生成するかが LSI 設計の課題となっている．

1.2 目的

以上の背景を踏まえ，本稿では Zynq-UltraScale+ MPSoC ZU7EV を利用した，FPGA による重力多体問題の数値アクセラレータを設計事例として，Vitis を用いた高位合成の性能予測を容易に行うための手法を提案することを最終目的とする．その前段階として，SDSoC と Vitis の違いについて調査する．高位合成で性能を出すためには様々な最適化オプションを適切に使用する必要があるが，ツールによってその挙動は変わるため，まずは SDSoC と Vitis との違いを把握する必要があるためである．ツールについて理解を深めたのちに，ツールの性能予測をし，それについて評価する．

1.3 本論文の構成

本論文の次章以下の構成を以下に示す．

1. はじめに
   本研究の背景と目的について記述する．
2. 既存研究
   本研究に関連する既存技術及び研究について説明し，本研究との差異及び参考にすることを述べる．
3. FPGA と開発環境
   本研究で用いる FPGA 評価ボードについて紹介する．また，開発環境について述べる．
4. 対象アプリケーションとシステム構成
   重力多体問題及び，本研究で実装したシステム概要について述べる．
5. ツール間比較
   重力多体問題の計算を SDSoC 及び Vitis で設計，性能比較する．
6. OpenCL での実装
   Vitis で複数パイプラインを実現するために OpenCL での実装，性能比較する．

7. 最適パラメータ推定手法
   Vitis を用いた設計で最適パラメータ推定手法を提案し，先行研究との比較，評価をする．

8. おわりに
   本研究のまとめと今後の課題と展望について述べる．

2　既存研究

本章では，SDSoC で及び Vitis を用いてアクセラレータ開発をした研究や高位合成で実装したアクセラレータの性能を予測した研究を紹介し，本研究との差異と新規性について説明する．

2.1　SDSoC によるアクセラレータ開発

2.1.1　An FPGA Realization of a Random Forest with k-Means Clustering Using a High-Level Synthesis Design

Akira JINGUJI らは，高位合成設計による k-Means クラスタリング付き Random Forest の FPGA 実現の研究を行った [4]．ランダムフォレスト (RF) とは，分類や回帰に用いられるアンサンブル機械学習アルゴリズムの一種である．ランダムフォレストは，ランダムに抽出されたデータから作成された複数の決定木から構成される．RF は他の機械学習アルゴリズムと比較して，単純かつ高速に学習・識別が可能である．様々な認識システムに応用され，広く利用されている．

ランダムフォレストは木ごとにアンバランストレースを行う必要があり，またすべての木に対して通信を行う必要があるため，GPU などの SIMD アーキテクチャには不向きである．FPGA を用いたアクセラレータも提案されているが，HDL 設計に基づく実装であった．また，FPGA を用いたアクセラレータも提案されているが，HDL 設計であるため，ソフトウエアによる実現に比べ，設計時間が長くなってしまう．前述した FPGA を用いたアクセラレータは，HDL 設計をベースとしているため，ソフトウエアで実現する場合に比べ，設計期間が長くなってしまう．

この研究では，さらにハードウェア量を削減するために，k-means クラスタリングを用いて決定木上の分岐ノードのコンパレータを共有化する．また，少数のハイパーパラメータでビットストリームを生成する krange ツールフローを開発する．提案ツールフローは高位合成設計に基づいているため，従来の HDL 設計と比較して短い設計時間で高性能な RF を得ることができる．この RF を Xilinx Inc. ZC702 評価ボードに実装した．CPU（Intel Xeon（R） E5607 Processor）および GPU（NVidia Geforce Titan）の実装と比較すると，性能面では FPGA 実装が CPU に比べて 8.4 倍，GPU に比べて 62.8 倍高速化された．消費電力効率については，FPGA 実装は CPU 実装の 7.8 倍，GPU 実装の 385.9 倍となっている (図 1).

本研究とは，SDSoC を用いて高位合成を行っている面で同様であり知見を得られるが，比較対象が CPU や GPU であるという点で異なる．

	GPU@86W 876 MHz Geforce Titan		CPU@13W 2.26 GHz Xeon (R) E5607		FPGA@14W 100 MHz Zynq7020	
Name	LPS	LPS/W	LPS	LPS/W	LPS	LPS/W
Arrhythmia	33.6	0.52	211.6	16.27	65.7	4.69
Dermatology	71.8	0.84	488.4	37.57	270.0	233.50
Ionosphere	82.1	0.95	595.9	45.84	3165.0	226.10
Iris	44.7	0.52	436.7	33.59	8087.0	577.60
Ratio	0.016	0.003	0.119	0.128	1.000	1.000

図 1: k-Means クラスタリング付き Random Forest の比較 (参考文献 [4] より引用)

2.1.2 Hot & Spicy: Improving Productivity with Python and HLS for FPGAs

Skalicky らは，Python アプリケーションに FPGA アクセラレータを統合するためのオープンソースのインフラストラクチャおよびツールスイートを発表した [5]．このツールスイートは，アクセラレータとその C/C++ベースのドライバのパッケージング，統合，バインディングを容易にし，Python アプリケーションから呼び出すことができるようにするものである．このツールは，以下のことが可能である．

1. Python 関数を HLS に適した C 関数に変換
2. Python C ラッパーバインディングを生成
3. FPGA EDA ツールフローを自動化
4. Python ソースコードを再ターゲットして加速ライブラリを使用

FPGA のエキスパートにとっては，追加のコンパイラや最適化を統合するためのフレームワークを提供することで，生産性の向上やフローの各段階における研究のサポートが可能になる．その他の人々にとっては，FPGA 上のアプリケーションを高速かつ一貫して高速化することができるが，FPGA 開発フローで高レベルの Python 抽象化をサポートするための設計原理とフローを説明されている．また，Python から C/C++ルーチンを呼び出す際のオーバーヘッドを評価している．最後に，Python 画像処理アプリケーションのカーネルを高速化した結果を示し，オリジナルの Python 実装に比べて 39,137 倍，高性能で手動最適化された OpenCV ライブラリ実装に比べて 6 倍の高速化が達成されている (図 2).

本研究とは，SDSoC を用いて高位合成を行っている面で同様であり知見を得られるが，比較対象が Python や OpenCV での実装であるという点で異なる．

Test Name	Performance	Speedup
Original Python	48.14 sec	1.0x
Refactored Python	139.28 sec	0.3x
Unoptimized HLS	58.68 ms	820.0x
Pipelined HLS	12.22 ms	3,939.0x
Partitioned HLS	1.23 ms	39,137.0x
OpenCV	7.19 ms	6,695.0x

図 2: Python,HLS,OpenCV での実装の性能比較 (参考文献 [5] より引用)

2.2 Vitis AI によるアクセラレータ開発

2.2.1 FPGA Implementation of Object Detection Accelerator Based on Vitis-AI

Jin Wang らは，Vitis-AI により物体検出アクセラレータの FPGA 実装を行った [6]. YOLOv3 の登場により，小さなターゲットの検出が可能になった．YOLO ネットワーク自体の特性上，YOLOv3 ネットワークは演算能力とメモリ帯域幅に対する要求が非常に高く，通常は専用のハードウェアアクセラレーションプラットフォーム上に配置する必要がある．FPGA は論理的に再構成可能なハードウェアチップで，性能と消費電力の面で大きな利点があるため，深層畳み込みネットワークを展開するのに適している．

この研究では，AXI バス ARM+FPGA アーキテクチャに基づく再構成可能な YOLOv3 FPGA ハードウェアアクセラレータを提案している．YOLOv3 ネットワークは Vitis AI を通じて定量化し，モデル圧縮やデータ前処理などの一連の処理により，アクセラレータチップや外部ストレージのアクセス時間を節約することが出来る．また，パイプライン演算により，FPGA の高スループットを実現した．GPU による YOLOv3 モデルの実装と比較すると，FPGA による YOLOv3 アクセラレータのハードウェア実装は消費電力が少なく，高いスループットを実現できている (図 3).

本研究とは，Vitis 用いて高位合成を行っている面で同様であり知見を得られるが，比較対象が GPU であるという点で異なる．

	FPGA (Xilinx ZCU104)	GPU
Platform	ZYNQ UltraScale+	GeForce GTX1080
Frquency (MHz)	300	10240
Precision	INT8	FP32
GOFs	5.5	2.45
FPS	84.5518 (Single thread) 206.701 (Multiple thread)	31.7
Power (W)	25	126
Energy Efficiency (FPS/W)	3.38	0.26

図 3: 物体検出アクセラレータの FPGA と GPU の比較 (参考文献 [6] より引用)

2.2.2 FA-LAMP: FPGA-Accelerated Learned Approximate Matrix Profile for Time Series Similarity Prediction

Kalantar らは，時系列類似性予測のための FPGA による加速学習型近似行列プロファイルである FA-LAMP を発表した [7]．低価格のセンサーや IoT（Internet-of-Things）の普及に伴い，データの生成速度は現在のインフラの計算能力やストレージ能力をはるかに超えてる．このようなデータの多くは時系列データであり，この 10 年，時系列アーカイブの構築や，データ処理のための新しい分析手法の開発・展開に関心が高まっている．一般的には，時系列データの様々な部分集合や部分配列に複数の類似性検索機構を適用して，繰り返しパターンや異常値を特定する方法が用いられてたが，これらのアプローチは計算量が多いため，電力制限のある今日の組み込み CPU とは相性が悪いとされている．

FA-LAMP は，学習型近似行列プロファイル（LAMP）アルゴリズムを FPGA で高速化したもので，リアルタイムでサンプリングされたストリーミングデータと学習に用いた代表的な時系列データセットの相関関係を予測する．また，分類や異常検出などの時系列解析問題のリアルタイムソリューションとして適している．さらに，加速された計算を IoT センサーのできるだけ近くに統合する仕組みを提供することで，事後分析のためのデータの送信やクラウドへの保存を不要にしている．

LAMP と FA-LAMP の中核には，予測を行うために CNN（Convolution Neural Networks）が採用されている．この目的のために構築された商業的にサポートされている最先端のフレームワーク，すなわち Xilinx Deep Learning Processor Unit（DPU）オーバーレイと Vitis AI 開発環境を使用する場合に，FPGA 上に CNN を展開する際の課題と限界を調査している．また，DPU のいくつかの技術的な限界を明らかにするとともに，DPU オーバーレイに独自の最適化 IP ブロックアクセラレータを取り付けることによって，これらの限界を克服するメカニズムを提供している．低コストな Xilinx Ultra96-V2 FPGA を用いて FA-LAMP を評価し，Raspberry Pi 3 上で動作するプロトタイプの LAMP 展開と比較して，性能とエネルギーが 1 桁以上向上していることを実証している (図 4).

本研究とは，Vitis 用いて高位合成を行っている面で同様であり知見を得られるが，比較対象が Raspberry 上での実装であるという点で異なる．

	Raspberry Pi 3	DPU + ARM	DPU + IP ultra_fast	DPU + IP fastexp_512
Inf. Rate (Hz)	1.4K	12.1K	15.0K	14.2K
Energy (J)	105.8	7.2	6.7	9.1

図 4: Raspberry Pi3 との性能，エネルギーの比較 (参考文献 [7] より引用)

2.3 高位合成でのアクセラレータ開発の性能予測

2.3.1 Performance assessment of FPGAs as HPC accelerators using the FPGA Empirical Roofline

Calore らは，FPGA Empirical Roofline を用いた HPC アクセラレータとしての FPGA の性能評価を行った [8]．現在，HPC システムにおいてハードウェアアクセラレータは非常に一般的であり，GPU はその中で主要な役割を果たしている．さらに最近では，特定のワークロードを高速化するために FPGA が一部のデータセンターで採用され始め，近い将来，汎用 HPC システムでも使用されるようになると予想される．FPGA は，いくつかの応用分野で興味深い高速化を提供することがすでに知られているが，典型的な HPC ワークロードのコンテキストで期待される性能を推定することは，簡単ではない．

これを容易にするため，HPC アプリケーションのハードウェアアクセラレータとして使用した場合の FPGA の計算スループットとメモリバンド幅を経験的に推定できるベンチマークツール，FPGA Empirical Roofline (FER) を提案している．FER は，広く知られている Roofline モデルを理論的基盤として，FPGA の演算スループットとバンド幅の上限を測定することができ，高位合成ツールで開発された関数カーネルの演算強度による性能を推定することができる．2 つの異なる高位合成パラダイムを使用して FER を実装した．これは，FPGA をハードウェアアクセラレータとして利用する HPC アプリケーションを開発するための 2 つの有望なアプローチである OmpSs@FPGA と Xilinx Vitis ワークフローである FER ベンチマークの理論モデルとその実装の詳細について説明し，Xilinx Alveo U250 FPGA で測定した性能結果を示している (図 5).

本研究とは，FPGA での実装の性能予測をしている面で同様であり知見を得られるが，1 つのアプリケーションの最適パラメータを推定するのが本研究の目的であることから違いがある．

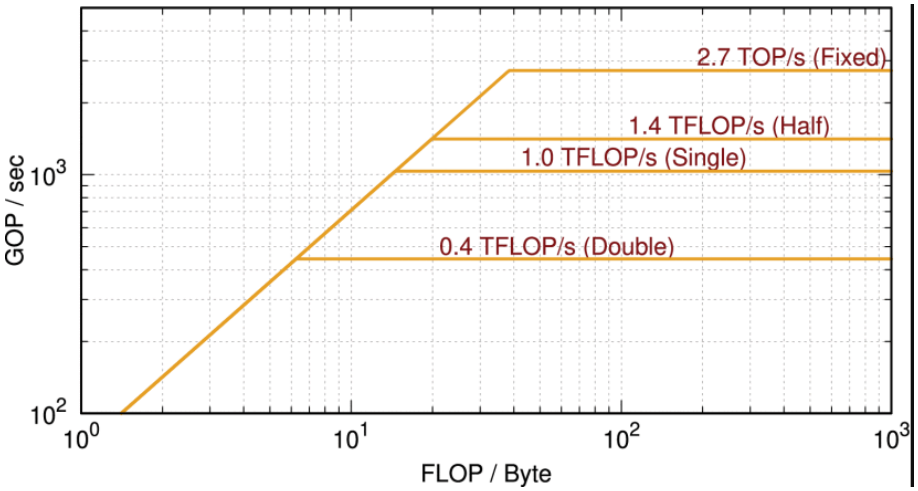

図 5: FER ベンチマークの性能結果 (参考文献 [8] より引用)

2.3.2 Resource and Performance Estimation for CNN Models using Machine Learning

Shahshahani らは，機械学習を用いた CNN モデルのリソースと性能の見積もりを行った [9]．FPGA（Field-Programmable Gate Array）ベースのハードウェアアクセラレータは，再構成性，性能，適応性，および優れたエネルギー効率を提供する．畳み込みニューラルネットワーク（CNN）ベースの推論システムの大部分は，PyTorch，Tensor Flow などの標準化されたフレームワークを使用して開発されている．これらの Python または Python に似たモデルは，FPGA 上にマッピングしてアクセラレータを構築することが出来る．FPGA 上にデザインを移植するためのマッピングフレームワークにより，CNN モデルを C/C++ や OpenCL などの高位言語に変換し，高位合成などの標準ツールが FPGA 上のモデルのマッピングを容易に行えるようになっている．FPGA ベースのアクセラレータの論理使用率と性能は，CNN ネットワークのパラメータ，アーキテクチャの選択（データフロー，パイプラインなど）によって左右される．

この研究では，スケーラブルな多層 CNN ハードウェアアクセラレータを Vitis 2020 HLS ツールでモデル化している．性能とハードウェアリソースの早期評価により，時間のかかる高位合成や FPGA の物理デザインマッピングを実行する前に，最適な CNN ネットワークを選択することが出来る．CNN の Python デザイン記述から論理使用率と計算時間を推定するための様々な機械学習（ML）モデルを提示している．その結果，HLS 合成を実行する前に，様々な多層 CNN ネットワーク (RFR(Random Forest Regression)，MLP(Multi-Layer Perceptron)，GBR(Gradient Boosting Regressor)) の性能とリソース使用量を短時間で正確

に推定することに成功している (図 6).

本研究とは，FPGA での実装の性能予測をしている面で同様であり知見を得られるが，複数アプリケーションの簡単な実装の性能を推定しているという点で異なる．

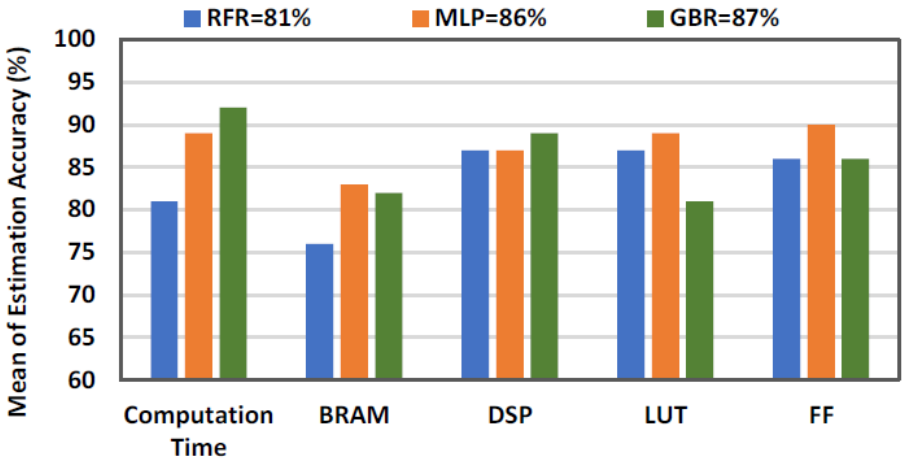

図 6: RFR，MLP，GBR でのネットワーク性能とリソース使用量 (参考文献 [9] より引用)

3 FPGA と開発環境

本章では，本研究の開発プラットフォームとなる Zynq-UltraScale+ MPSoC EV，高位合成ツールの SDSoC と Vitis について紹介する．

3.1 FPGA

FPGA（Field Programmable Gate Array）はプログラマブルロジックデバイスの一種で，通常の専用ハードウェアとは異なり製造後に回路の構成を変更することができる集積回路である．回路の論理記述をユーザが変更するには，VerilogHDL や VHDL などのハード ウェア記述言語が用いられる．FPGA は何度でも論理の再構成が可能であるため，製造後に論理を変更できない ASIC(Application Specific Integrated Circuit) の動作検証や頻繁に回路を変更する必要のある製品などに使用されている．本研究では，Zynq-UltraScale+ MPSoC デバイス [10] を搭載した FPGA 評価ボードである ZCU104 を使用した．

3.2 ZCU104

今回の開発で使用した FPGA 評価ボードは Xilinx 社の ZCU104 であり，FPGA としては Zynq-UltraScale+ MPSoC ZU7EV(表 1) が搭載されている．この FPGA の特徴は最大 1.5GHZ で動作するクワッドコア Arm Cortex-A53 プラットフォームである．加えて，デュアルコア Cortex-R5 リアルタイムプロセッサ，Mali-400MP 2 GPU，および 16nm FinFET+ プログラマブルロジックと統合されている (図 7).

表 1: Zynq UltraScale+ XCZU7EV-2FFVC1156 MPSoC

システムロジックセル (K)	504
メモリ	38Mb
DSP スライス	1,728
ビデオコーデックユニット	1
最大 I/O ピン数	464

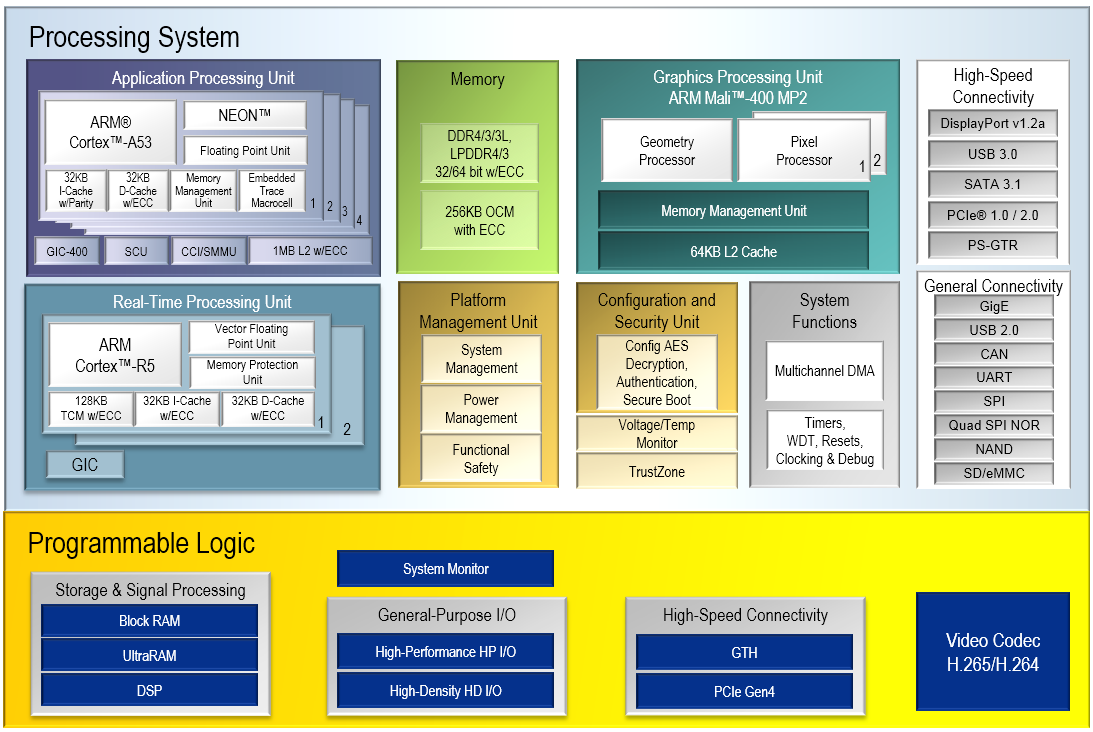

図 7: Zynq-UltraScale+ MPSoC EV の内部構造 (参考文献 [10] より引用)

3.3　高位合成ツール

3.3.1　高位合成

FPGA を含めるディジタル回路設計は，古典的には AND,OR のゲートレベルで回路を記述していたが，近年では HDL を利用したそれより抽象度の高い RTL レベルで回路を設計することが主流になった．RTL 記述は人手による詳細な最適化が可能であるが，開発工数が多く人為的なミスの混入の可能性から逃れられない．そこで，高位合成と呼ばれる，高級言語から HDL(Hardware Description Language) を生成，ツールによってはゲートレベルのネットリストまでを生成して回路を設計まで行う技術が研究されてきた．しかし，ソフトウェアの記述がそのまま回路として合成できるわけではなく，またその性能は記述の形式と合成ツールに支配される．高位合成による回路の最適化を行うには，CPU 向けのソフトウェアコードから機能面では問題とならないタイミング制約や回路資源を考慮した動作となるよう記述内容を検討・変更することが好ましい．

高位合成では一般に変数はレジスタ，配列はメモリ，関数は回路モジュールとしてインスタンス化され，逐次実行，ループ，分岐といった制御はステートマシンとして実現される．変数，配列の bit 幅によって回路規模と演算精度のトレードオフが成立する．また，依存関係がない演算や制御は自動的に処理系が並列動作する回路を生成するが，処理が複雑な場合は依存関係のない記述でも処理系がそのように判断できない場合もある．そこで，多くの高位合成の処理系はディレクティブや拡張命令によって，依存関係の明示的な指定や，モジュールのパイプライン化や並列化を記述することができる．

3.3.2　SDSoC

SDSoC は，高位合成のために Xilinx 社が提供する Eclipse IDE ベースの Zynq SoC/MPSoC エンデベッドアプリケーションシステムの統合開発環境である．SDSoC では C/C++/OpenCL

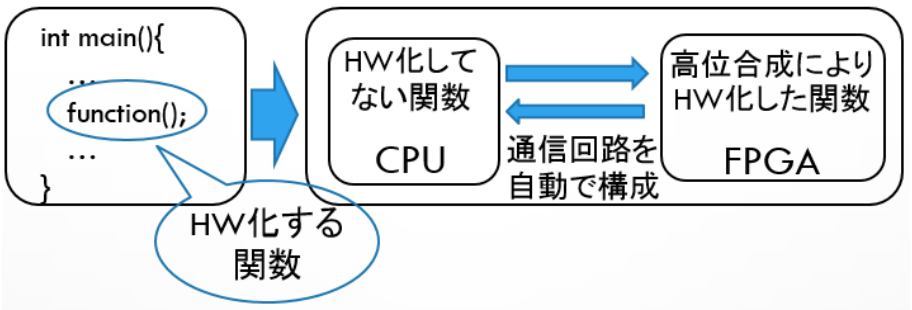

図 8: 通信回路も自動生成する高位合成

での開発をカバーし，１つのプログラムコードから FPGA アクセラレーションを含めたソフトウェアアプリケーションを実現できる．このときデータモーションネットワークと呼ばれる CPU-FPGA 間の通信インターフェイスも自動的に生成する点が SDSoC の特徴である (図 8)．本研究で使用した SDSoC のバージョンは 2019.1 である．

3.3.3 Vitis

Vitis は，Xilinx FPGA，SoC，Versal ACAP でエンベデットソフトウェアおよびアクセラレーションアプリケーションの統合開発環境である．Vitis では C/C++/Python/OpenCL での開発をカバーし，その他に RTL ベースアクセラレータと低レベルなランタイム API を使用して，より細かく実装を制御することも可能である．高位合成をする際には，Vivado でクロックや AXI ポートなど FPGA 内のどの部分を使ってよいか定義したり，petalinux でアプリケーションを動かすための Linux 環境を作成したりと高位合成前の作業が多い点が SDSoC との違いである (図 9)．SDSoC では Linux を用いない Baremetal 環境も使える．使用した Vitis のバージョンは 2020.2 である．

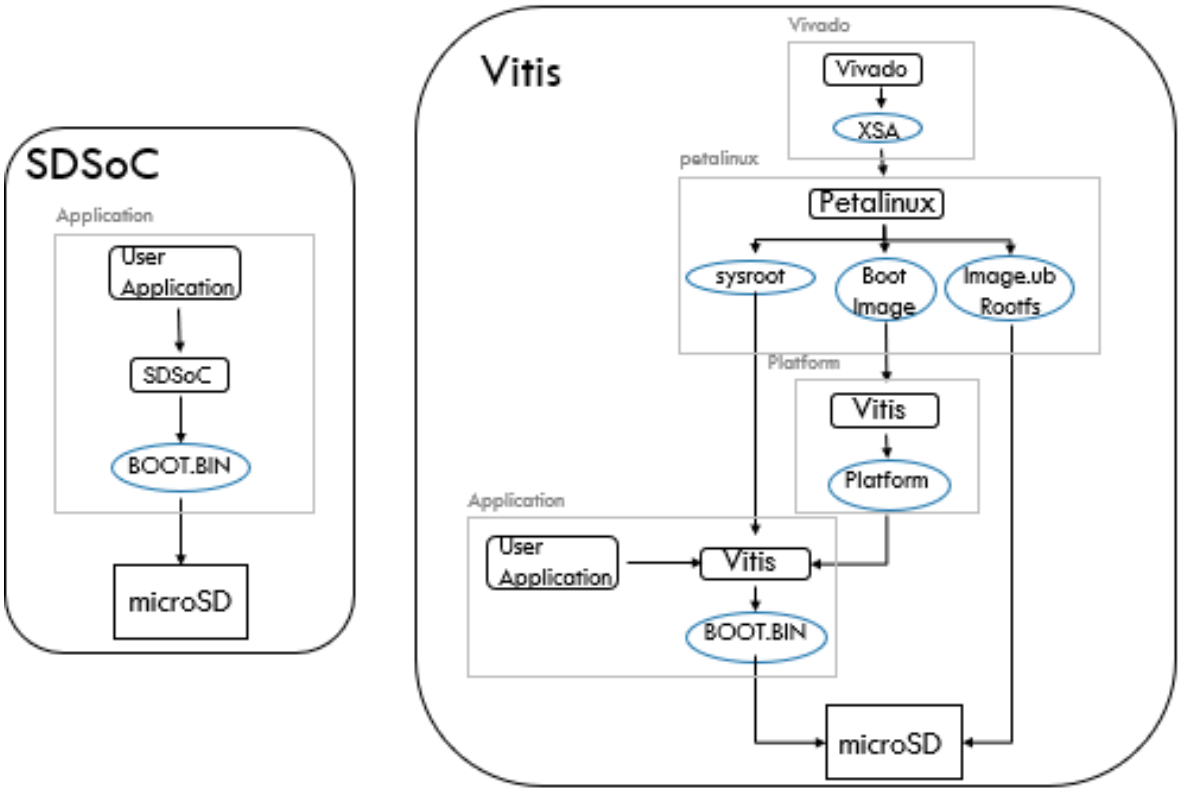

図 9: SDSoC と Vitis の設計フロー

4　重力多体問題

本研究では FPGA による数値アクセラレータの対象として重力多体問題を扱う．重力多体問題は何もない宇宙空間に質点をばら撒いたとき，その後の運動が万有引力によってどう変化するものかを問うものである．三体問題以上は解析的な算出が困難であることから，差分法による計算技術を用いる．重力多体問題計算では質点に働く力の計算が最も大変であるため，本研究では力の計算を行う部分のみハードウェア化する．

質点 j が質点 i に及ぼす万有引力の x 成分を $f_{x_{ij}}$・m_i とすれば，

$$f_{x_{ij}} m_i = G m_i m_j \frac{x_i - x_j}{r_{ij}^3}. \tag{1}$$

ここでは，r_{ij} は質点 i と質点 j の距離，m_i, m_jはそれぞれ質点 i と質点 j の質量，G は万有引力定数である．これより，単位質量あたりの質点 i が受ける万有引力の x 成分の合計 f_{x_i} は，

$$f_{x_i} = \sum_{j=0}^{N} f_{x_{ij}}. \tag{2}$$

ここで N は質点の数である．これを y 方向と z 方向においても同様にして求めることで重力多体問題を計算することが出来る (リスト 1, リスト 2).

本論文では計算速度として Gflops の単位を使っているが， 1 ペアの粒子間の力の計算に 38 演算を要していると仮定して flops に換算している．リスト 1 のように平方根や逆数の演算が あってそれらは加算や乗算より多くの時間がかかることから重 力多体問題分野での慣例として 38 の数字を使用している．

リスト 1: 重力多体問題の計算プログラム

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#define EPS2 (0.03f*0.03f)
extern void float_force_optimized_sub(int n, float posf[][4], float forcef[][4])
{
  int i,j,k;
  float dr[3],r_1,dtmp,r2,fi[4],sqrtfeps2=1.0f/sqrtf(EPS2);
  for(i=0;i<n;i++){
    for(k=0;k<4;k++) fi[k]=0.0f;
    for(j=0;j<n;j++){
#pragma HLS PIPELINE
      r2=EPS2;
      for(k=0;k<3;k++){
        dr[k] = posf[i][k]- posf[j][k];
        r2 += dr[k] * dr[k];
      }
      r_1=1.0f/sqrtf(r2);
      dtmp=posf[j][3]*r_1;
      fi[3]+=dtmp;
      dtmp*=r_1*r_1;
      for(k=0;k<3;k++){
        fi[k] -= dtmp * dr[k];
      }
    }
    fi[3]-=posf[i][3]*sqrtfeps2;
    for(k=0;k<4;k++){
      forcef[i][k]=fi[k]*posf[i][3];
    }
  }
}
```

リスト 2: 非同期実行無しのホストプログラム

```
void float_force_optimized(int n, double pos[][4], double force[][4])
{
  int i,j,k;
  float (*posf)[4],(*forcef_1)[4]/*,(*forcef_2)[4]*/;
  posf=(float (*)[4])malloc(sizeof(float)*n*4);
  forcef_1=(float (*)[4])malloc(sizeof(float)*n*4);
  for(j=0;j<n;j++) for(k=0;k<4;k++) posf[j][k]=(float)pos[j][k];
  float_force_optimized_sub(n,posf,forcef_1);
  for(i=0;i<n;i++) for(k=0;k<4;k++) force[i][k]=(double)forcef_1[i][k];
  free(posf);free(forcef_1);
}
```

5　ツール間比較

本章では，SDSoC 及び Vitis を用いて重力多体問題を計算する数値アクセラレータを実装し，いくつかの項目でツール間の性能を比較した．

5.1　ソースコードの修正

まず基本的な方針として，なるべくソースコードには修正を加えないでどの程度の高速化が出来るかを比較する．今回ハードウェア化したい関数はリスト 1 の float_force_optimized_sub である．SDSoC の場合は開発ツール内で関数を選んで pull-down メニューからハードウェア化を選ぶだけであるが，Vitis では一旦別ファイルに分けてから指定する必要があった．次に SDSoC ではリスト 2 の 5,6 行目の malloc 関数を sds_malloc に，10 行目の free を sds_free に変更する必要があるが，Vitis ではその必要がない．高速化のためにハードウェア化する部分はパイプライン処理した方がよい．SDSoC ではリスト 1 の 12 行目のように pragma で指定しないと駄目だが，Vitis では特に pragma を挿入しなくても自動的にパイプライン化された．図 10 は SDSoC でこの pragma を挿入するかしないかで計算性能を比較したものである．粒子数がある程度あればパイプライン化した方が遥かに性能が出ることが分かる．

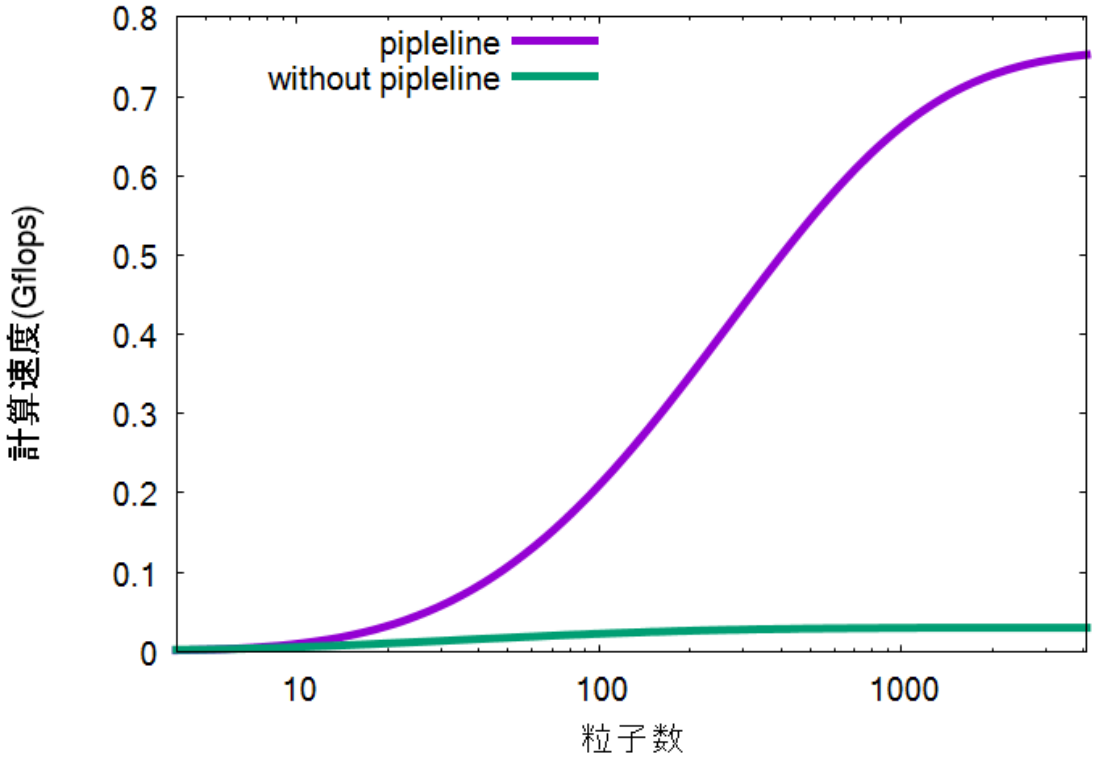

図 10: パイプライン処理ありとなしの計算速度の比較

5.2 パイプラインの計算性能

ここでは作成したハードウェアパイプラインの最大性能を測定する．重力多体問題での力の計算ルーチンはリスト 1 のように粒子数 N の二乗の計算が必要である一方ハードウェアとの通信量は粒子数に比例する．つまり粒子数が大きければほとんどの処理時間はパイプラインでの計算に費やされるため，粒子数が多い時の全体の処理時間からパイプラインのピークの計算性能を見積もることが出来る．今 FPGA での専用ハードウェアによる 1 回の float_force_optimized_sub の処理時間を T_{fpga} とした時,

$$T_{fpga} = t_{fpga} \times N^2 \tag{3}$$

と表すことが出来る．ここで t_{fpga} は 1 ペアの粒子の力の計算にかかる時間である．粒子数が 4,096，パイプラインのクロック周波数が 150MHz の時の値は表 2 の様になった．t_{fpga} が小さいことから SDSoC の方がパイプラインの性能が高い．

表 2: 処理時間の実測値

	SDSoC	Vitis
t_{fpga}	3.4×10^{-8}	6.7×10^{-8}
t_{cpu}	2.7×10^{-7}	2.0×10^{-7}
t_{band}	2.6×10^{-9}	7.9×10^{-11}
t_{lat}	4.8×10^{-4}	0

5.3 通信性能

専用パイプラインはアプリケーションプログラムからデータを受け取り処理結果を返す必要がある．1 回の float_force_optimized_sub の呼び出しに対応する通信時間を T_{comm} とすると，

$$T_{comm} = t_{band} \times 32 \times N + 2 + t_{lat} \tag{4}$$

と表すことが出来る．ここで t_{band} は 1byte を転送するのにかかる時間，t_{lat} は一回の転送を始める際のセットアップ時間である．1 粒子の座標は 16byte，1 粒子に働く力も 16byte のデータが必要であり，座標と力の 2 回の転送を必要とするため式 (4) のようになる．

転送速度を測るルーチンを作成して計測したところ，t_{band} , t_{lat} は表 2 のようになった．Vitis での t_{lat} は非常に小さいマイナスの値が算出されたため 0 とみなしている．最大転送速度は Vitis が速く，t_{lat} も Vitis が小さい．このため転送量が少ない時には SDSoC の方が転送に時間がかかる．

5.4 全体性能

リスト 2 のサブルーチンを 1 回処理する時間 T は，

$$T = T_{fpga} + T_{cpu} + T_{comm} \tag{5}$$

と表せる．ここで T_{cpu} は

$$T_{cpu} = t_{cpu} \times N \tag{6}$$

と表せる．

ここで，T_{fpga} は，ハードウェア化した関数内で 11 回ループ (10 回は無駄なループ) した場合とそうでない場合の T の時間差を 10 で割り，これを何回か繰り返し平均をとることで求めた．$T_{cpu} + T_{DM}$ は，ハードウェア化した関数内でほとんど何もせず return する

プログラムに修正した場合の T を測定し，これを何回か繰り返し平均をとる事で求めた．T_{comm} は，T_{fpga} を測定したときと同じように無駄なソケット通信を行わせたときの T の増分から求めた．t_{cpu} は表 2 のようになった．式 (3)-(6) から計算速度を求めたものが図 11 である．曲線は T を，点は実測値を意味する．SDSoC も Vitis も粒子数が多くなるほど計算速度が速くなるが，最大性能では SDSoC が勝ることが分かる．一方粒子数が少ない時は転送速度が Vitis の方が速いため Vitis の計算速度の方が速い．

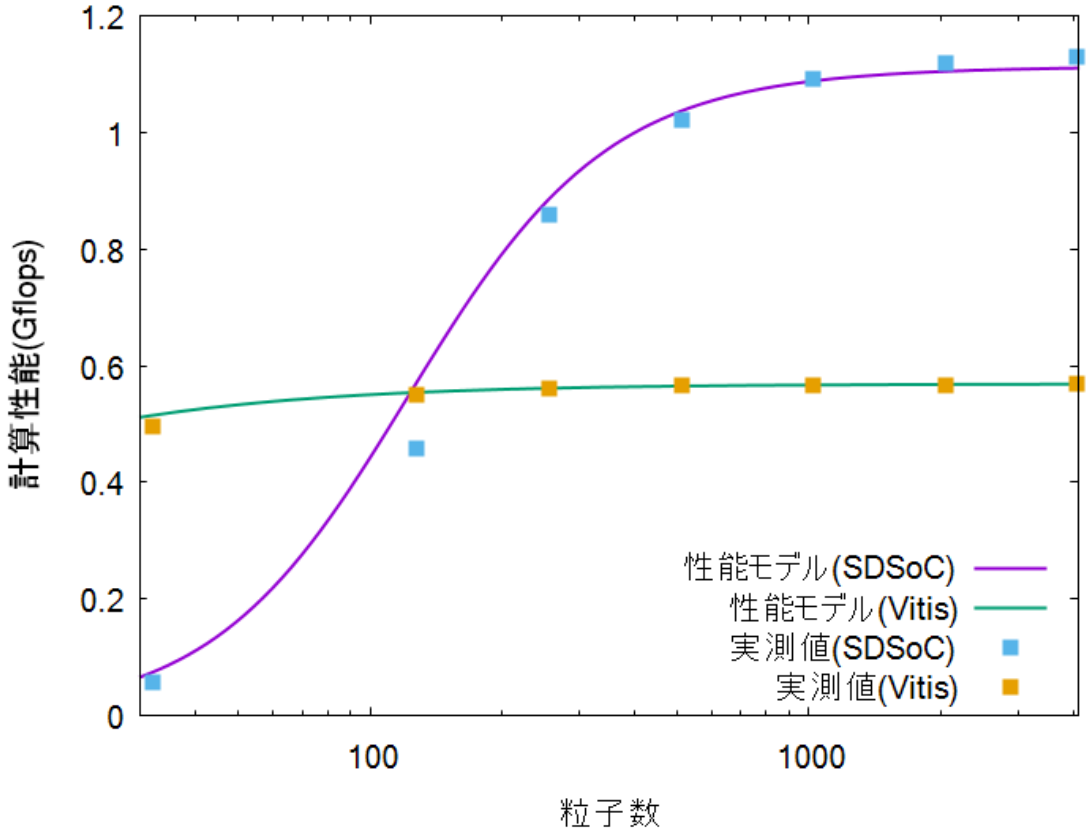

図 11: SDSoC と Vitis のパイプラインの計算速度の比較

5.5 クロック周波数の変更

計算パイプラインのクロック周波数を変えることが出来る．SDSoC では pull-down メニューから選べるが，Vitis ではコンパイルオプションとしてテキスト入力する必要がある．クロックを変えた際の $N = 4,096$ の時の計算速度は表 3 のようになった．Vitis ではクロック周波数が変更されているものの計算速度が向上していない．interval は 1 ペアの粒子への力の計算に何クロックかかるかを意味するが，クロック周波数が上がっても interval が同じだけ増えた場合は計算速度は変わらない．SDSoC の方ではクロック周波数の向上ほどは interval が大きくなっていないので計算速度が向上している．

5.6 リソース

クロック周波数が 150MHz の際に使用した回路資源の量を比較した (表 4)．BRAM と LUT は Vitis の方が少なく，DSP と FF は Vitis の方が多い．Vitis では回路資源の使用量を確認する際，新たに LUTRAM の値を確認できるようになっている．

表 3: クロック周波数による計算速度の違い

	目標周波数 (MHz)	最大周波数 (MHz)	interval	計算速度 (Gflops)
SDSoC	75	70.6	4	0.71
	150	133.4	5	1.14
	300	274.3	8	1.41
Vitis	75	72.4	4	0.57
	150	139.9	4	0.57
	300	262.7	7	0.57

表 4: 回路資源の比較

	BRAM	DSP	FF	LUT	LUTRUM
SDSoC	39	10	11,970	8,825	n/a
Vitis	8	13	8,848	12,320	600
上限値	624	1,728	460,800	230,400	110,460

5.7 複数パイプライン

先行研究では，重力多体問題の計算部分を複数の計算ユニットに分けて並列に計算することで計算性能の向上を図った．SDSoC も Vitis も HLS の機能でループアンローリングを行って並列処理をすることが可能であるが，ここでは（データ転送を含んで）関数全体を複数並列化したい場合を考える．SDSoC では関数呼び出しの部分に特有の SDS プラグマを使用することで非同期で計算ユニットを呼び出し並列に計算させることが容易であった (リスト 3)．ここで sub1 内では前半だけの i について計算し，sub2 内では後半だけの i について計算している．

一方，Vitis では SDS プラグマは使用できないので OpenCl を用いてアウトオブオーダー実行を指定することで並列処理を行うことが出来る (リスト 4)．2 行目のように，コマンドキューを作成するときに CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE を引数に設定する．コードを大きく書き換える必要があるため，プラグマのように普通のコンパイラでコンパイルするだけで元のコードが走るような便利さは無くなる．

リスト 3: SDS プラグマの使用例

```
#pragma SDS async(1)
float_force_optimized_sub1(n,posf,forcef_1);
#pragma SDS async(2)
float_force_optimized_sub2(n,posf,forcef_2);
#pragma SDS wait(1)
#pragma SDS wait(2)
```

リスト 4: OpenCl での並列実行の例

```
LOG("clCreateCommandQueue\n");
cmd_queue = clCreateCommandQueue(context, device_id,CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE
    , &err);
if (cmd_queue == NULL) {
  fprintf(stderr, "can't create command queue: %d\n", err);
  return 1;
}
```

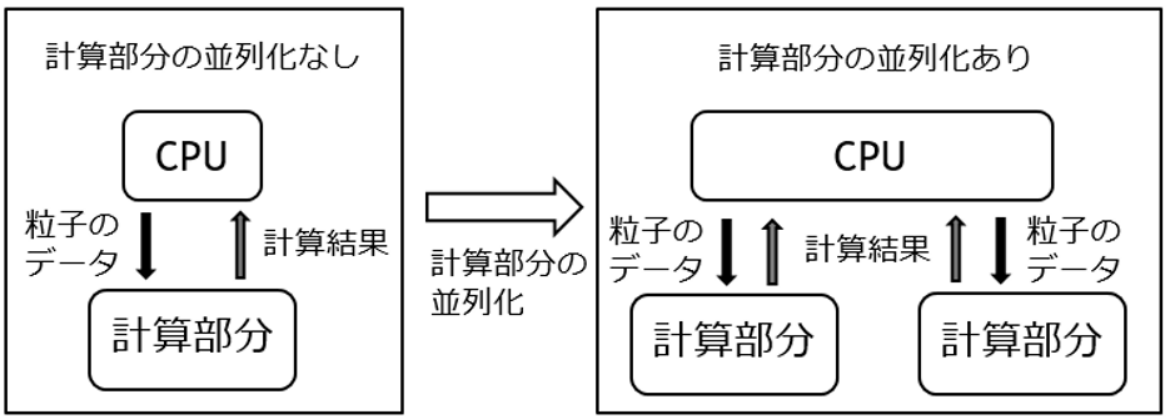

図 12: 複数パイプラインのイメージ図

6 OpenCL での実装との比較

前章では，ソースコードを大きく変更することはなく，SDSoC と Vitis で実装した場合のツール間の挙動を比較した．前章での実験の結果，複数パイプラインを実装するためには OpenCL での記述が必要であることが分かった．

本章では，Vitis で OpenCL を用いてコードを大きく変更し，実装した場合について計算性能の比較を行う．

6.1 OpenCL での実装

OpenCL を用いて実装したホストプログラムの解説を行う．以下に処理の流れと実行する OpenCL の関数を表 5 に示す．

表 5: OpenCL での実装で使用する関数

実行すること	使用する関数
プラットフォーム (cl_platform_id) の取得	clGetPlatformIDs
デバイス (cl_device_id) の取得	clGetDeviceIDs
コンテキスト (cl_context) の作成	clCreateContext
コマンドキュー (cl_command_queue) の作成	clCreateCommandQueue
プログラム (cl_program) のロード	clCreateProgramWithBinary
カーネル (cl_kernel) の取得	clCreateKernel
メモリの準備とバッファ(cl_mem) の作成	clCreateBuffer
カーネル引数の設定	clSetKernelArg
入力データの転送要求の投入	clEnqueueMigrateMemObjects
カーネルの実行要求の投入	clEnqueueTask
出力データの転送要求の投入	clEnqueueMigrateMemObjects
完了待ち合わせ	clFinish
リソース解放	clReleaseMemObject

6.1.1 プラットフォーム (cl_platform_id) の取得とデバイス (cl_device_id) の取得

関数 clGetPlatformIDs は，1 つ目の引数でプラットフォームの数を決めて 2 つ目の引数にその情報を保存する．本研究のホストプログラムでは，プラットフォームは 1 種類であるとしている．プラットフォームが複数ある場合 (例えば FPGA と GPU など) は，複数取得して選択する必要がある．プラットフォームの選択は関数 clGetPlatformInfo を用いる (リスト 5).

6.1.2 デバイス (cl_device_id) の取得

関数 clGetDeviceIDs は，関数 clGetPlatformIDs と似たような使い方でデバイスを取得する．本研究のホストプログラムでは，デバイスは 1 台であるとしている．デバイスが複数ある場合は，複数取得して選択する必要がある．プラットフォームの選択は関数 clGetDeviceInfo を用いる (リスト 6).

リスト 5: 非同期実行無しのホストプログラム

```
1   err = clGetPlatformIDs(1, &platform_id, NULL);
2   if (err != CL_SUCCESS) {
3     fprintf(stderr, "can't get platform ids: %d\n", err);
4     return 1;
5   }
6
7   err = clGetPlatformInfo(platform_id, CL_PLATFORM_VENDOR, 100, (void *)platform_vendor,
        NULL);
8   if (err != CL_SUCCESS) {
9     fprintf(stderr, "can't get platform info: %d\n", err);
10    return 1;
11  }
12  if (strcmp("Xilinx", platform_vendor)) {
13    fprintf(stderr, "vender is not xilinx.\n");
14    return 1;
15  }
```

リスト 6: 非同期実行無しのホストプログラム

```
1     err = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_ACCELERATOR,1, &device_id, NULL);
2     if (err != CL_SUCCESS) {
3       fprintf(stderr, "can't get device ids: %d\n", err);
4       return 1;
5     }
```

6.1.3 コンテキスト (cl_context) の作成

関数 clCreateContext は，OpenCL コンテキストを作成するために用いる．コンテキストは，コマンドキュー，メモリ，プログラム，カーネルなどのオブジェクトを扱うために用いられる (リスト 7).

6.1.4 コマンドキュー (cl_command_queue) の作成

関数 clCreateCommandQueue は，OpenCL コマンドキューを作成するために用いる．第 5 章で述べたが，複数パイプラインの並列実行をするためにアウトオブオーダー実行を行うために必要である．このコマンドキューを作成する関数を使用するために全体を OpenCL を用いて書き変える必要があった．アウトオブオーダー実行のためには 3 つ目の引数に CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE を加える必要がある (リスト 8).

リスト 7: コンテキスト (cl_context) の作成プログラム

```
1     context = clCreateContext(NULL, 1, &device_id, NULL, NULL, &err);
2     if (context == NULL) {
3       fprintf(stderr, "can't create context: %d\n", err);
4       return 1;
5     }
```

リスト 8: コマンドキュー (cl_command_queue) の作成プログラム

```
1     cmd_queue = clCreateCommandQueue(context, device_id,CL_QUEUE_PROFILING_ENABLE, &err);
2     if (cmd_queue == NULL) {
3       fprintf(stderr, "can't create command queue: %d\n", err);
4       return 1;
5     }
```

6.1.5 プログラム (cl_program) のロード

プログラム (cl_program) の取得は，関数 clCreateProgramWithBinary を使用する．clCreateProgramWithBinary には，xclbin ファイルの内容を渡す必要がある．xclbin ファイルとは FPGA をプログラムするのに必要なバイナリファイルのことである (リスト 9).

6.1.6 カーネル (cl_kernel) の取得

Vitis では HW 化する部分を一旦別ファイルに分けてから指定する必要があるが，その別ファイルから作成した FPGA カーネルを取得するために関数 clCreateKernel を用いる (リスト 10).

リスト 9: プログラム (cl_program) のロードプログラム

```
1     program = clCreateProgramWithBinary(context, 1, &device_id, &xclbin_size, (const
          unsigned char **)&xclbin, NULL, &err);
2     if (program == NULL) {
3       fprintf(stderr, "can't create program: %d\n", err);
4       return 1;
5     }
6
7     if (munmap(xclbin, xclbin_size) != 0) {
8       perror("munmap");
9       return 1;
10    }
11    close(fd);
```

リスト 10: カーネル (cl_kernel) の取得プログラム

```
1     kernel = clCreateKernel(program, "vadd", &err);
2     if (kernel == NULL) {
3       fprintf(stderr, "can't create kernel: %d\n", err);
4       return 1;
5     }
```

6.1.7　メモリの準備とバッファ(cl_mem) の作成〜リソース解放

第 4 章のリスト 2 で OpenCL を用いない場合のホストプログラムを紹介している．この部分を OpenCL を用いて記述した (リスト 11).

関数 clCreateBuffer でプロセスメモリと関連付けたバッファの作成を行っている．入力用 (posf) と出力用 (forcef) で一部引数が異なる．バッファは，デバイス側のメモリを管理するためのリソースだが，ここではプロセスメモリと紐づけている．

カーネル引数の設定には，関数 clSetKernelArg を用いる．また，データ転送及びカーネル実行要求には関数 clEnqueueMigrateMemObjects と関数 clEnqueueTask を用いる．これも同じ関数を入力用と出力用で引数を変えて用いる．OpenCL を用いない場合は，計算部分の関数に粒子のデータを引数で設定すれば出来る動作だが，OpenCL を用いる場合は，1 つの引数に対して 1 つの関数で設定し，データ転送を行わなければならない．

関数 clFinish は，コマンドキューに投入した要求の完了を待つものであり，完了したら関数 clReleaseMemObject でリソースを開放する．

リスト 11: メモリの準備とバッファ(cl_mem) の作成〜リソース解放プログラム

```
1     posf = clCreateBuffer(context,  CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR, sizeof(
          cl_float) * n * 4, x, &err);
2     if (posf == NULL) {
3       fprintf(stderr, "can't create buffer posf: %d\n", err);
4       return 1;
5     }
6     forcef = clCreateBuffer(context,  CL_MEM_WRITE_ONLY | CL_MEM_USE_HOST_PTR, sizeof(
          cl_float) * n * 4, a2, &err);
7     if (forcef == NULL) {
8       fprintf(stderr, "can't create buffer forcef: %d\n", err);
9       return 1;
10    }
11    err = clSetKernelArg(kernel, 0, sizeof(cl_int), &n);
12    if (err != CL_SUCCESS) {
13      fprintf(stderr, "can't set kernel arg 0: %d\n", err);
14      return 1;
15    }
16    err = clSetKernelArg(kernel, 1, sizeof(cl_mem), &posf);
17    if (err != CL_SUCCESS) {
18      fprintf(stderr, "can't set kernel arg 1: %d\n", err);
19      return 1;
20    }
21    err = clSetKernelArg(kernel, 2, sizeof(cl_mem), &forcef);
22    if (err != CL_SUCCESS) {
23      fprintf(stderr, "can't set kernel arg 2: %d\n", err);
24      return 1;
25    }
26
27    err = clEnqueueMigrateMemObjects(cmd_queue, 1, &posf,
28          0, 0, NULL, NULL);
29    if (err != CL_SUCCESS) {
30      fprintf(stderr, "can't migrate mem object posf: %d\n", err);
31      return 1;
32    }
33
34    err = clEnqueueTask(cmd_queue, kernel, 0, NULL, NULL);
35    if (err != CL_SUCCESS) {
36      fprintf(stderr, "can't enqueue kernel: %d\n", err);
37      return 1;
38    }
39
40    err = clEnqueueMigrateMemObjects(cmd_queue, 1, &forcef,
41            CL_MIGRATE_MEM_OBJECT_HOST, 0, NULL, NULL);
42    if (err != CL_SUCCESS) {
43      fprintf(stderr, "can't migrate mem object forcef: %d\n", err);
44      return 1;
45    }
46
47    err = clFinish(cmd_queue);
48    if (err != CL_SUCCESS) {
49      fprintf(stderr, "can't finish: %d\n", err);
50      return 1;
51    }
52
53    /* Freeing resouces to show examples. */
54    clReleaseMemObject(posf);
55    clReleaseMemObject(forcef);
```

6.2 性能モデル及び実測値

まずは OpenCL を用いて重力多体の計算を行うプログラムを実装する．この際，プラグマや計算全体の並列化など最適化手法は一切用いなかった場合での比較である．前章と同様にパイプラインの計算性能，通信性能，全体性能について述べる．

前章と同様にして t_{fpga}, t_{cpu}, t_{band}, t_{lat} を求めた (表 6)．この表より計算性能は OpenCL での実装をする前よりも向上し．通信性能は，逆に低下していることが分かった．

また全体性能は，前章の図 11 と比較したものを図 13 に示した．この図から OpenCL での実装は SDSoC での実装に近い挙動をすることが分かった．

表 6: 処理時間の実測値

	SDSoC	Vitis	Vitis(OpenCL)
t_{fpga}	3.4×10^{-8}	6.7×10^{-8}	5.7×10^{-8}
t_{cpu}	2.7×10^{-7}	2.0×10^{-7}	3.0×10^{-7}
t_{band}	2.6×10^{-9}	7.9×10^{-11}	1.5×10^{-9}
t_{lat}	4.8×10^{-4}	0	6.3×10^{-4}

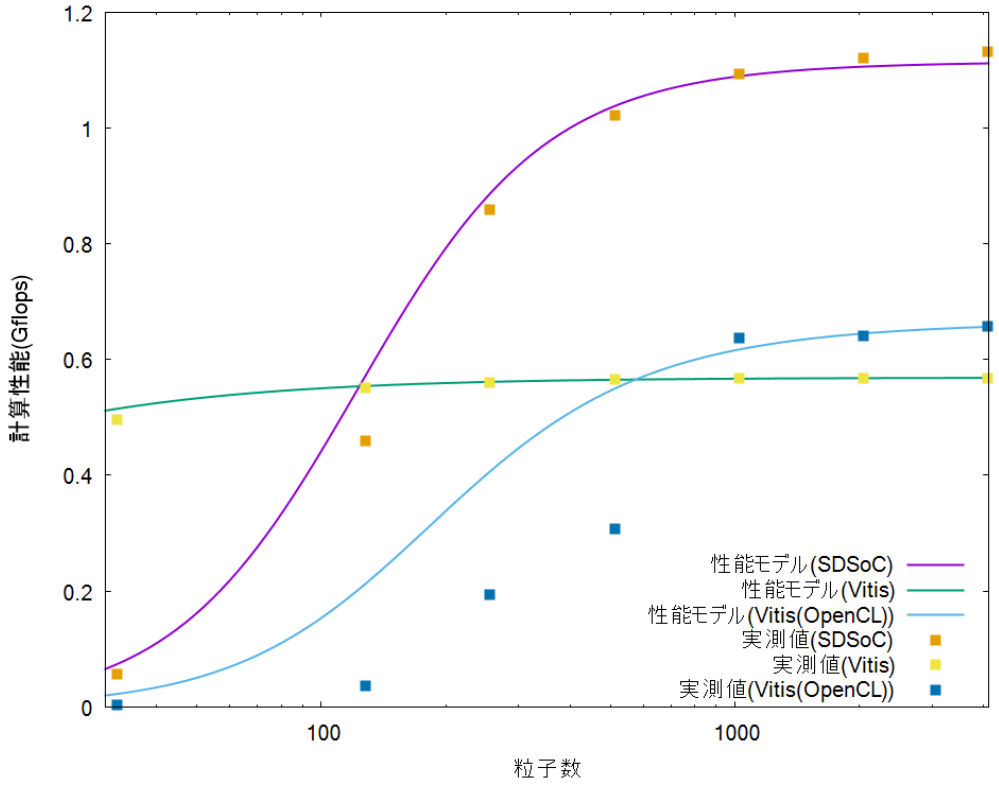

図 13: OpenCL 使用の実装と OpenCL 不使用の実装と SDSoC での実装の比較

6.3 クロック周波数の変更

前章と同様にクロック周波数を変更した際の計算速度を表 7 に示した．OpenCL を用いる前の実装ではクロック周波数を変更しても計算速度が向上しない若干不可思議な現象が起こっていたが OpenCL での実装ではクロック周波数の変更に応じて計算性能が変わっていることが分かる．しかし，周波数を大きくしても interval も増加して計算性能は，向上することがない．このアプリケーションの場合 Vitis での実装では 150MHz での性能が最大であると考察できる．

6.4 リソース

ここでも前章と同じようにクロック周波数が 150MHz の際の使用した回路資源の量を比較した (表 8)．BRAM と DSP と LUTRAM は同じ値だったが，FF と LUT は OpenCL での実装の方が多い．この差が計算性能の差に関わっていると考えられる．

表 7: クロック周波数による計算速度の違い

	目標周波数 (MHz)	最大周波数 (MHz)	interval	計算速度 (Gflops)
OpenCL 不使用	75	72.4	4	0.57
	150	139.9	4	0.57
	300	262.7	7	0.57
OpenCL 使用	75	72.4	4	0.35
	150	144.1	4	0.66
	300	218.1	7	0.67

表 8: 回路資源の比較

	BRAM	DSP	FF	LUT	LUTRUM
変更なし	8	13	8,848	12,320	600
OpenCL	8	13	9,253	13,021	600
上限値	624	1,728	460,800	230,400	110,460

6.5 プラグマによるループアンローリング

高位合成ツールでは，HLS プラグマによる計算性能向上を行うことが出来る．先行研究では，SDSoC においてプラグマによるループアンローリングを行ったが計算性能の向上にはつながっていない．図 9 に，プラグマを用いた場合の計算性能及び回路資源の比較を示した．これより，先行研究と同様にプラグマによるループアンローリングでは計算性能の向上が確認できなかった．

6.6 手動でのループアンローリング

前章で，プラグマを用いたループアンローリングを行ったが計算性能の向上には至らなかった．そこで，先行研究と同様に手動でループアンローリングを行った．まずは計算のプログラム内にある k の小さなループをすべて無くした．表 10 に，計算性能及び回路資源の比較を示した．この結果より，ほとんど回路資源の使用量は変わらずに大きく計算性能を引き上げることに成功した．その後，i のループにおいても手動でループアンローリングを行った．表 11，図 14 に，回路資源及び計算性能の比較を示した．これより，k のループを削除したときには大幅に向上が見られ, さらに i のループ内をループ展開すると計算性能はおよそ比例して向上することが分かった．しかし，回路資源にはまだ余裕があるがループ展開を 64 にするとコンパイルに失敗してしまった．

6.7 複数パイプライン

前章で述べた関数全体の複数並列化を行った．並列化を行うために計算部分のプログラム及びホストプログラムを変更した．ホストプログラム及び計算部分のプログラムでの主な変更点は,

1. ホストプログラムでは，計算部分を分割するために FPGA カーネルを複数作る必要があるので配列を用いて FPGA カーネルを作るように変更した．
2. 粒子間の計算を分割するために各カーネルの計算の初めを配列 ioffset を用いて表した．
3. 計算部分のプログラムでは，粒子の計算の初めを指定するための配列 ioffset を引数に追加した．

の 3 点である．変更したプログラムの部分をリスト 12，13 に示した．また，表 12 に並列化した数に応じた回路資源を，計算性能を図 15 示した．この結果より，先行研究と同様に並列数におよそ比例して計算性能は向上し，回路資源の使用量も増加することが示された．

表 9: プラグマを用いたループアンローリングによる計算性能及び回路資源の比較

	BRAM	DSP	FF	LUT	LUTRUM	計算速度
プラグマなし	8	13	12,735	9,147	923	0.66Gflops
プラグマあり	8	13	13,153	10,122	1,052	0.66Gflops
上限値	624	1,728	460,800	230,400	110,460	

表 10: k のループを削除したことによる計算性能及び回路資源の比較

	BRAM	DSP	FF	LUT	LUTRUM	計算速度
変更前	8	13	12,735	9,147	923	0.66Gflops
変更後	8	16	11,377	9,587	1,171	1.40Gflops
上限値	624	1,728	460,800	230,400	110,460	

表 11: 手動でのループアンローリングによる回路資源の比較

	BRAM	DSP	FF	LUT	LUTRUM	interval
ループ展開なし	4	16	11,377	9,587	1,171	4
2	8	16	13,219	10,070	1,110	5
4	8	20	19,634	14,551	1,303	5
8	16	40	25,286	20,293	1,788	5
16	16	77	42,975	35,439	2,662	5
32	16	151	77,995	64,098	4,355	5
64	-	-	-	-	-	-
上限値	624	1,728	460,800	230,400	110,460	-

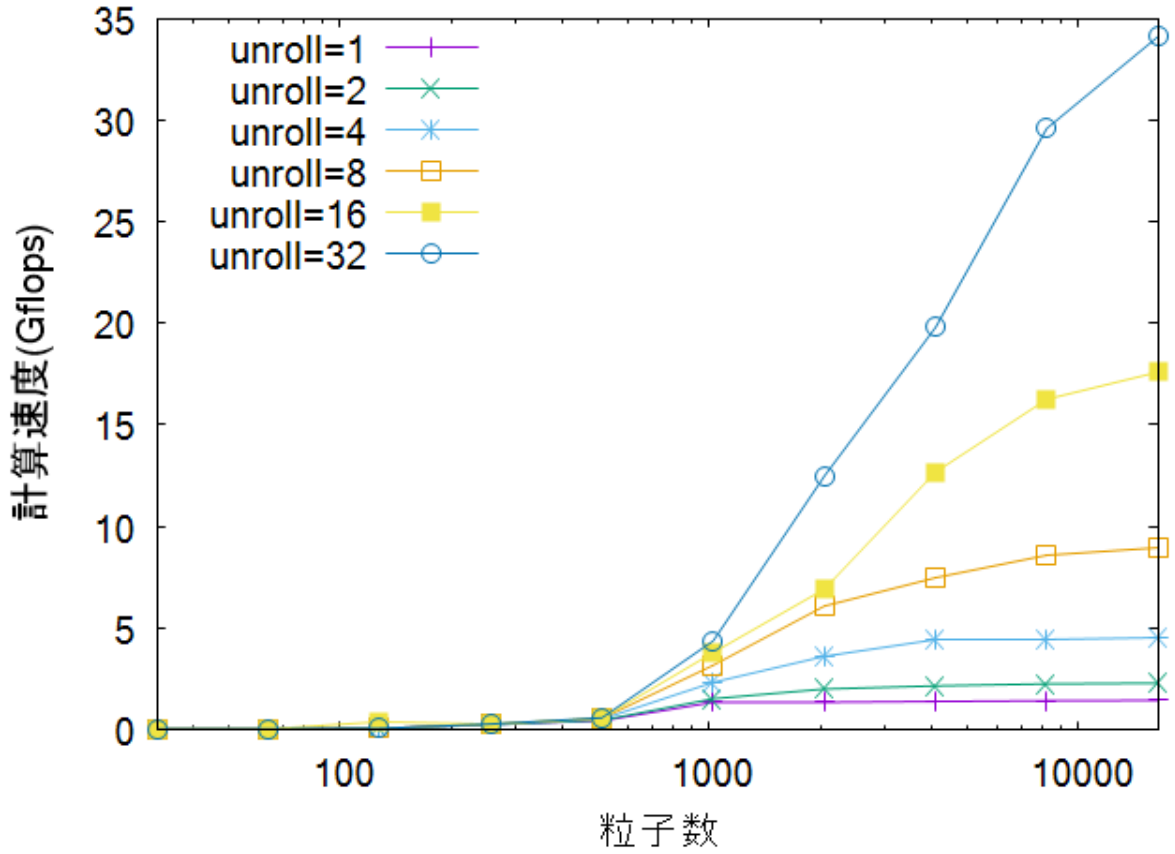

図 14: 手動でのループ展開による計算性能比較

リスト 12: 複数パイプラインのためのホストプログラム

```
    for(i=0;i<CU;i++){
      kernel[i] = clCreateKernel(program, "gravity", &err);
      if (kernel == NULL) {
        fprintf(stderr, "can't create kernel %d : %d\n", i,  err);
        return 1;
      }
    }


    for(i=0;i<CU;i++){
      //LOG("clSetKernelArg 0\n");
      err = clSetKernelArg(kernel[i], 0, sizeof(cl_int), &ioffset[i]);
      if (err != CL_SUCCESS) {
        fprintf(stderr, "can't set kernel arg 0: %d\n", err);
        return 1;
      }
      //LOG("clSetKernelArg 1\n");
      err = clSetKernelArg(kernel[i], 1, sizeof(cl_int), &n_cu);
      if (err != CL_SUCCESS) {
        fprintf(stderr, "can't set kernel arg 1: %d\n", err);
        return 1;
      }
      //LOG("clSetKernelArg 2\n");
      err = clSetKernelArg(kernel[i], 2, sizeof(cl_int), &n);
      if (err != CL_SUCCESS) {
        fprintf(stderr, "can't set kernel arg 2: %d\n", err);
        return 1;
      }
      //LOG("clSetKernelArg 3\n");
      err = clSetKernelArg(kernel[i], 3, sizeof(cl_mem), &posf);
      if (err != CL_SUCCESS) {
        fprintf(stderr, "can't set kernel arg 3: %d\n", err);
        return 1;
      }
      //LOG("clSetKernelArg 4\n");
      err = clSetKernelArg(kernel[i], 4, sizeof(cl_mem), &forcef);
      if (err != CL_SUCCESS) {
        fprintf(stderr, "can't set kernel arg 4: %d\n", err);
        return 1;
      }
      //LOG("clEnqueueTask\n");
      err = clEnqueueTask(cmd_queue, kernel[i], 0, NULL, NULL);
      if (err != CL_SUCCESS) {
        fprintf(stderr, "can't enqueue kernel: %d\n", err);
        return 1;
      }
    }
```

リスト 13: 複数パイプラインのための重力多体問題の計算プログラム

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#define EPS2    (0.03f*0.03f)

void vadd(int ioffset, int ni, int nj,
    float *posf, float *forcef)
{
  int i,j,k;
  float dr[3],r_1,dtmp,r2,fi[4],sqrtfeps2=1.0f/sqrtf(EPS2);

  for(i=ioffset;i<ioffset+ni;i++){
    for(k=0;k<4;k++) fi[k]=0.0f;
    for(j=0;j<nj;j++){
  #pragma HLS pipeline II=1
      r2=EPS2;
      for(k=0;k<3;k++){
        dr[k]=posf[i*4+k]-posf[j*4+k];
        r2+=dr[k]*dr[k];
      }
      r_1=1.0f/sqrtf(r2);
      dtmp=posf[j*4+3]*r_1;
      fi[3]+=dtmp;
      dtmp*=r_1*r_1;
      for(k=0;k<3;k++){
        fi[k]-=dtmp*dr[k];
      }
    }
    fi[3]-=posf[i*4+3]*sqrtfeps2;
    for(k=0;k<4;k++){
      forcef[i*4+k]=fi[k]*posf[i*4+3];
    }
  }
}
```

表 12: 計算全体の並列化による回路資源の比較

計算ユニット	BRAM	DSP	FF	LUT	LUTRUM
1	8	16	11377	9587	1171
2	43	32	25566	19313	2519
4	77	64	45654	35727	4791
7	137	112	77137	60806	8297
8	154	128	87040	68895	9432
上限値	624	1,728	460,800	230,400	110,460

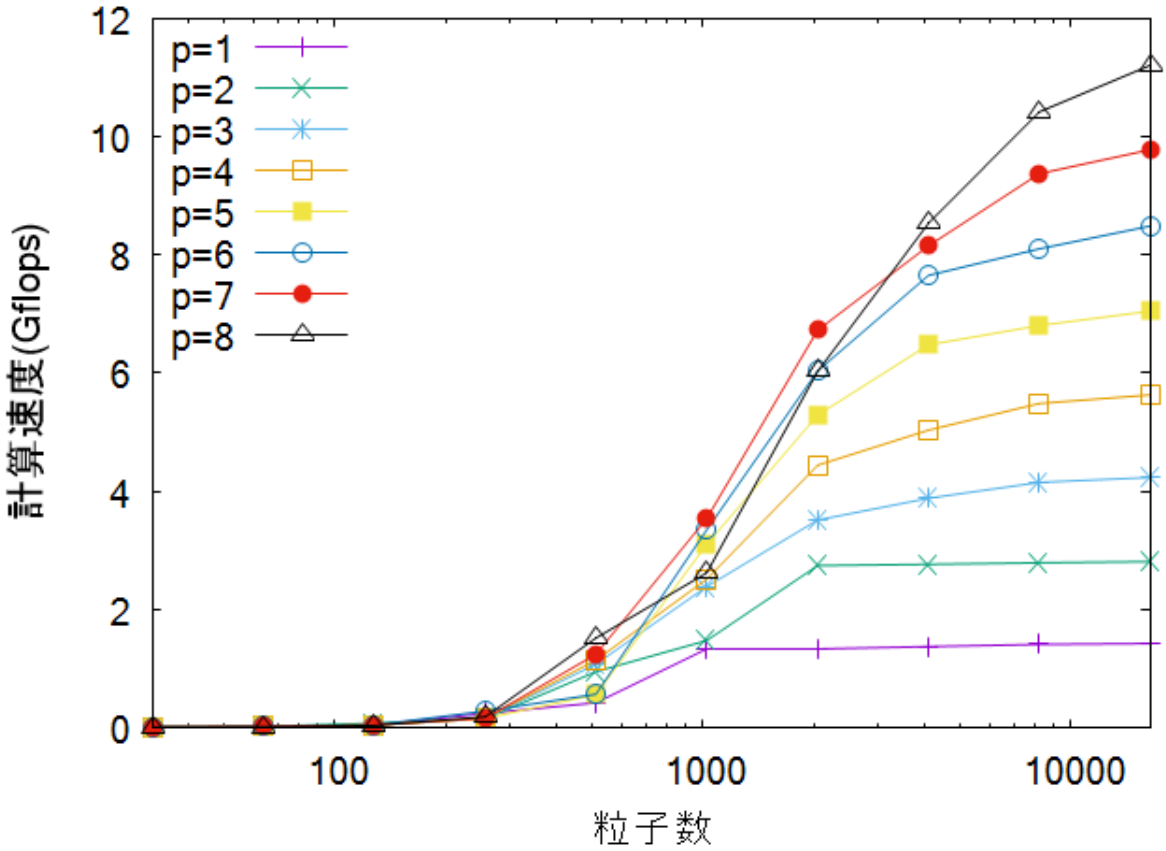

図 15: 複数パイプラインの計算性能比較

7　最適パラメータの推定

第 5,6 章では，最適化手法として主にクロック周波数の変更，手動でのループ展開，複数パイプラインでの計算を行い比較した．その際，最適化手法を重ねていくうえで BRAM と LUT がボトルネックになる可能性があると考えられる．本章ではまず BRAM と LUT の使用量を推定する性能モデルを作成し，それをもとに最適パラメータ推定手法を提案し，いくつかの条件に応じた最適パラメータを求めた．

7.1　回路資源の推定

第 6 章の表 11 及び表 12 よりループ展開と複数パイプラインはその数が増えるとおよそ比例して増加していることが分かる．BRAM の総数 B_{total} は，次のように推定できる．

$$B_{total} = B_{pipe} p_{dma} + B_{other} \tag{7}$$

先行研究では N_{local} というローカルメモリのサイズを変更して推定しているが本研究では alligned_alloc を用いているためローカルサイズの変更はしないものとする．B_{pipe} はデータ転送用の BRAM の数と粒子の位置と力を格納するための BRAM の数の合計で，表 12 から計算ユニットの数と BRAM の値を用いて差分から求めた．B_{other} はパイプライン以外に必要な BRAM の数である．先ほど求めた $B_{mem}N_{local} + B_{dma}$ を用いて求めた．また，L_{total} は次のように推定できる．

$$L_{total} = L_{pipe} p_{dma} + L_{other} \tag{8}$$

ここで，L_{pipe} は DMA コントローラと同様にパイプラインの LUT ユニット数で，表 12 から計算ユニットの数と BRAM の値を用いて差分から求めた．L_{total} はその他のロジック用の LUT で，先ほど求めた L_{pipe} を用いて求めた．表 13 に求めたパラメータを示す．また同様にしてループ展開の場合も調査した．このとき BRAM はボトルネックにならないと推測できるので LUT のみ調査した．

また，回路資源量の推定モデルと実測値の比較を図 16,17 に示した．

表 13: 回路資源に関するパラメータ

	複数パイプライン	ループ展開
B_{pipe}	17	-
B_{other}	18	-
L_{pipe}	8,089	1,791
L_{other}	4,183	6,786

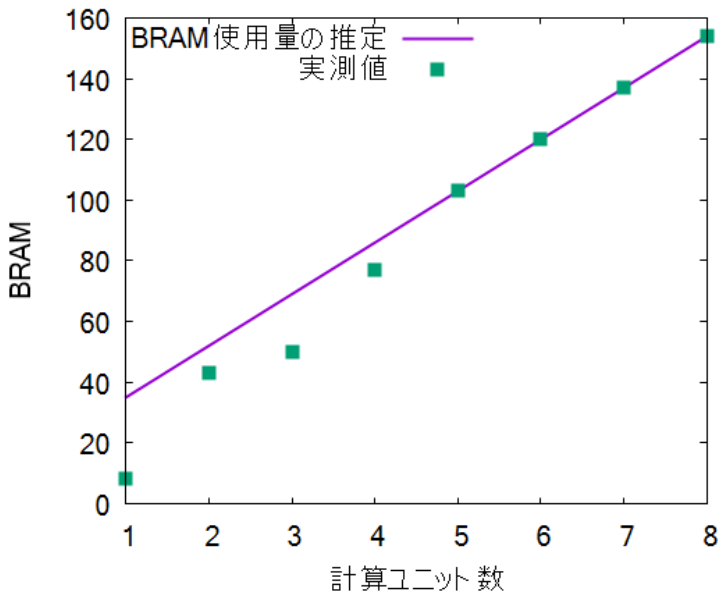

図 16: BRAM の使用量の推定

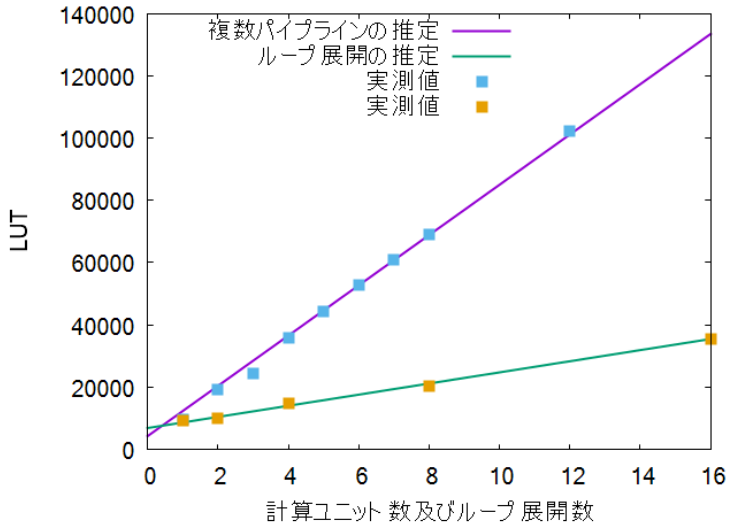

図 17: LUT の使用量の推定

7.2 最適パラメータ推定

前章までの調査をもとに最適パラメータを推定する．本研究では最適なパラメータとしてループ展開を行うのか複数パイプラインでの計算を行うのか重力多体計算の粒子数に応じて推定する．第 6 章の表 11 よりループ展開を行うと計算の inerval は 4 から 5 になるので単純にループ展開の方が 1 つのパイプラインの性能が 4/5 になる．したがって，ループ展開で 16 並列で計算した場合と計算部分を 12 個の計算ユニットに分割した場合のピーク性能はおよそ一致すると考えられる．図 18 にこの 2 つの場合の性能モデル及び実測値を示した．この図より粒子数を多くしてピーク性能での比較ではループ展開での実装の方がより性能がいいと推定されるが，粒子数がおよそ 2000 以下の場合では複数パイプラインでの実装の方が性能がいいと推定できた．これは通信時間の全体に占める割合が大きいときに通信部分も分割し並列で通信を行えるという点で優位に立つことが出来ていると考えられる．

実際に，実測値と比較すると計算性能に関しては粒子数が 128 から 2048 のところでは性能モデルと離れてしまっているが粒子数が少ないところでは，複数パイプラインの方が優れているという推定は正しいことが確認できた．

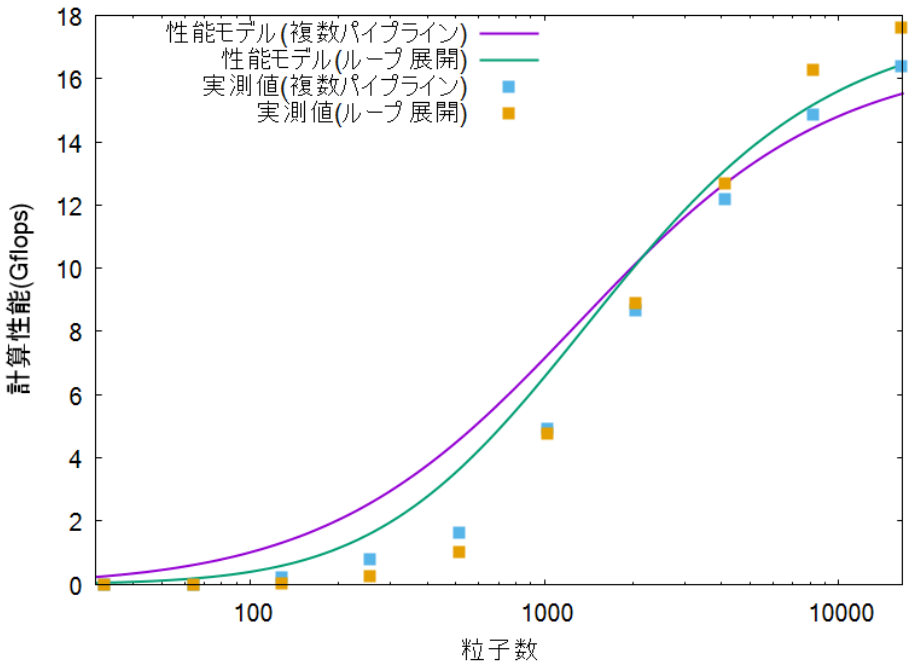

図 18: 複数パイプライン及びループ展開の最大性能間の比較

8　おわりに

本章では，FPGA ＋ CPU プラットフォームにおける重力多体問題アプリケーション実装に関する高位合成ツール間比較および最適化手法に関して，本研究の取り組みをまとめ，今後の展望に関して述べる．

8.1　まとめ

本稿では，重力多体問題を例として 2 つの高位合成ツールによって実装，比較し評価した．Vitis での実装は，SDSoC とパイプライン性能及び通信性能で違う特徴を示し，またクロック周波数の変更では，性能が変わらないなど不可思議な挙動も見受けられた．しかし，OpenCL を用いて実装をすると SDSoC と似たような挙動を示すことが分かった．また，Vitis での実装は SDSoC に比べて回路資源の使用量が低くなることが分かった．

また性能予測について，複数パイプラインでの並列化とループ展開での並列化のどちらが最適となるか性能モデルから推定した．限定的ではあるが複数パイプラインの利点である通信部分も含めた並列化から通信時間の全体に占める割合が大きいときには複数パイプラインでの実装をするとよいという予測が得られた．

8.2　今後の課題と展望

8.2.1　複数の手法を合わせての実装

本研究では，最後の推定としてループ展開での並列化と複数パイプラインでの並列化の 2 つを別で用いてアクセラレータを実装し，計算性能を推定した．しかし，主に回路資源に関して制約がなければ 2 つを組み合わせてより計算性能を引き上げることが出来ると推定できる．そこで回路資源の使用に制約をかけることでその中で最適なパラメータを推定することが今後の研究の課題として挙げられる．

8.2.2　別の手法での比較

本研究では，最終目的となる最適パラメータの推定を行うために最適化手法をいくつかに絞って比較，検討を行った．しかし，アクセラレータの性能向上のためにはまだいくつもの手法が残されているのでその比較，検討からさらにツール間の特徴を捉え，より最適な実装に近づくことが出来ると考えられる．

8.2.3　最適パラメータ推定手法の改良

本研究では，複数パイプラインでの並列化ととループ展開での並列化のどちらが最適となるか性能モデルから推定した．しかし，かなり限定的な場面でのみ有効なものとなっ

てしまった．より汎用性のある手法を提案するために研究を続けていく必要がある．

8.2.4　他のアプリケーションへの応用

本研究では，計算性能の比較のために適していると考えられる重力多体問題を例として実装を行った．しかし，FPGA デバイスで実装することが適しているアプリケーションは他にもあるため，そのアプリケーションでも比較，検討することでよりツールの優位性を発見できる可能性があると考えられる．

参考文献

[1] Xilinx 社，”SDSoC 開発環境”.
https://japan.xilinx.com/products/design-tools/software-zone/sdsoc.html

[2] Xilinx 社，”Vitis 開発環境”.
https://japan.xilinx.com/products/design-tools/vitis.html

[3] 高村 政孝，高位合成を用いた FPGA の開発.
https://www.oki.com/jp/otr/2015/n225/pdf/otr225_r09.pdf

[4] Akira JINGUJI, Shimpei SATO, and Hiroki NAKAHARA, An FPGA Realization of a Random Forest with k-Means Clustering Using a High-Level Synthesis Design, IEICE Transactions on Information and Systems, E101.D(2), pp. 354-362, 2018

[5] Sam Skalicky, Joshua Monson, Andrew Schmidt, and Matthew French, Hot & Spicy: Improving Productivity with Python and HLS for FPGAs, 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp.85-92,2018

[6] Amin Kalantar, Zachary Zimmerman, and Philip Brisk, FA-LAMP: FPGA-Accelerated Learned Approximate Matrix Profile for Time Series Similarity Prediction, 2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 40-49, 2021

[7] Jin Wang and Shenshen Gu, FPGA Implementation of Object Detection Accelerator Based on Vitis-AI, 2021 11th International Conference on Information Science and Technology (ICIST), pp. 571-577, 2021

[8] Masoud Shahshahani and Dinesh Bhatia, Resource and Performance Estimation for CNN Models using Machine Learning, 2021 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), pp. 43-48, 2021

[9] Enrico Calore and Sebastiano Fabio Schifano, Performance assessment of FPGAs as HPC accelerators using the FPGA Empirical Roofline, 2021 31st International Conference on Field-Programmable Logic and Applications (FPL), pp. 83-90, 2021

[10] Xilinx 社，”Zynq UltraScale+ MPSoC”.
https://japan.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html

[11] Xilinx 社，”SDSoC 環境最適化ガイド”.
https://www.xilinx.com/support/documentation/sw_manuals_j/xilinx2017_1/ug1235-sdsoc-optimization-guide.pdf

[12] Xilinx 社，”SDSoC 開発環境ヘルプ”.
https://japan.xilinx.com/html_docs/xilinx2019_1/sdsoc_doc/ysq1526001978941.html

[13] Xilinx 社，Zynq UltraScale+ MPSoC ZCU104 評価キット
https://japan.xilinx.com/products/boards-and-kits/zcu104.html

[14] Xilinx 社，”Vitis 統合ソフトウェア開発プラットフォーム 2020.2”.
https://japan.xilinx.com/html_docs/xilinx2020_2/vitis_doc/hlspragmas.html

謝辞

本研究に取り組むにあたって，きめ細やかな指導やアドバイスをいただいた指導教員である成見 哲教授や，副指導教員の吉永 努教授に深謝致します.