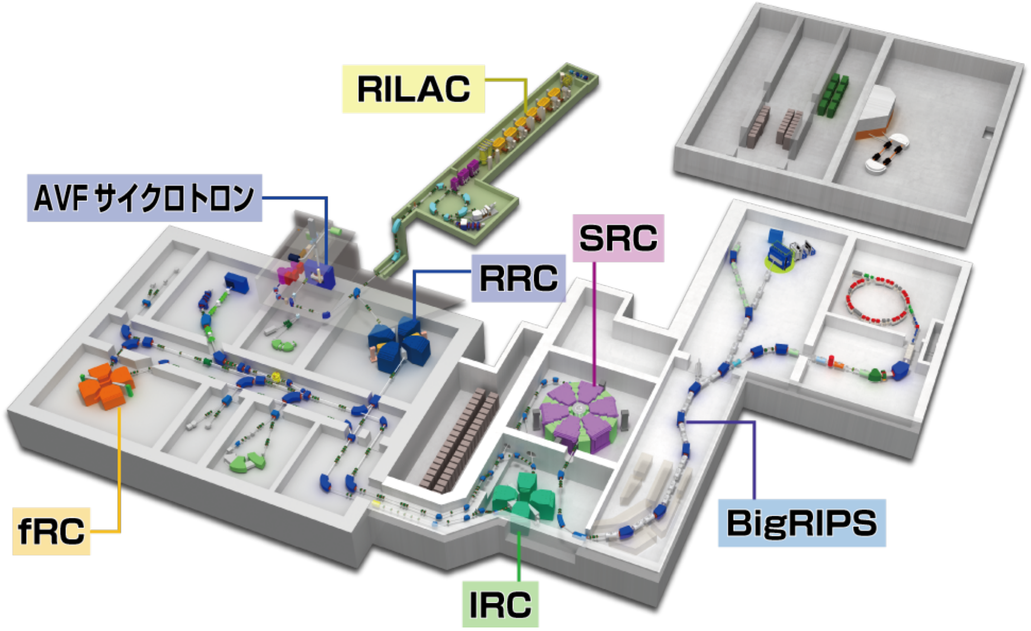
Alveo を用いた RIBF DAQ データ解析

ー 未経験から始めるハードウェアアクセラレーション

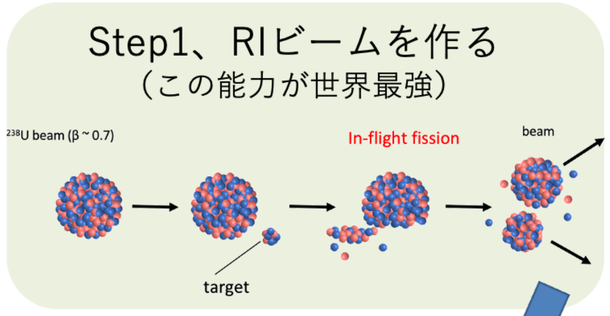
Y. Ichinohe (RIKEN Nishina Center)
2023.11.21

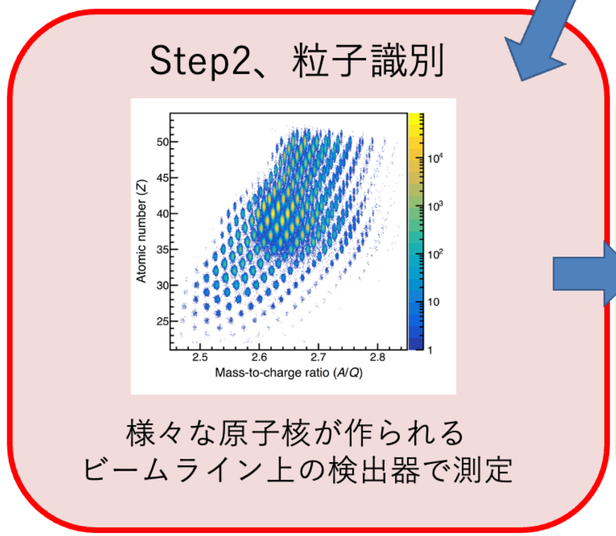
計測システム研究会 2023 @ RCNP

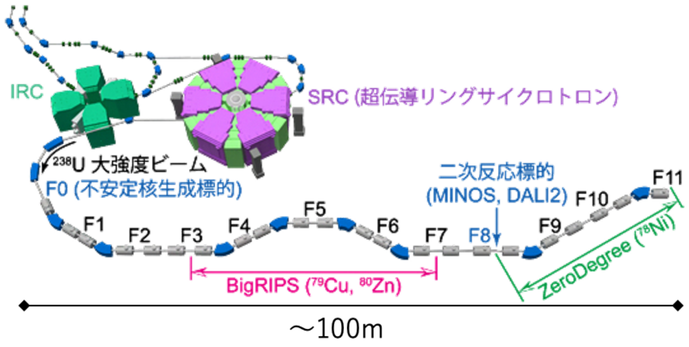
理研 RIBF (RI Beam Factory)

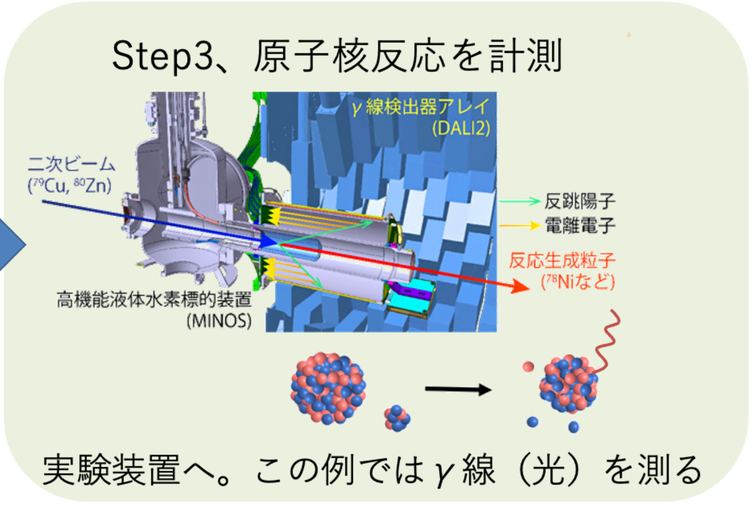


- 線形加速器+サイクロトロンの組み合わせで RI の二次ビームを生成
  - 二次ビームの intensity: << 1e7 cps
- BigRIPS (超伝導 RI ビーム生成分離装置) で RI を ID
- メイン検出器で物理測定









BigRIPS

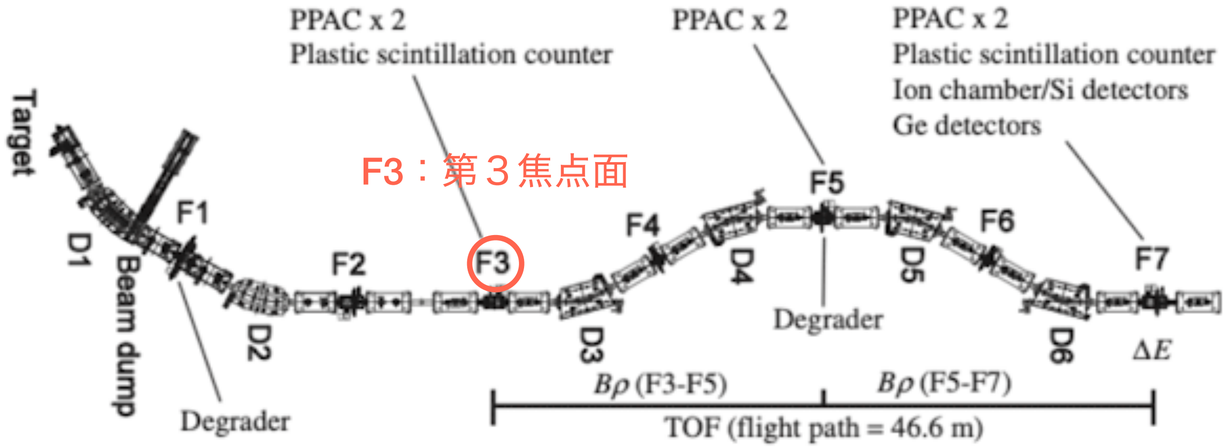
ビーム上に PID のための数種類の検出器

- プラシンチ →TOF
- PPAC → 荷電粒子の位置と向き
- イオンチェンバー → エネルギーロス

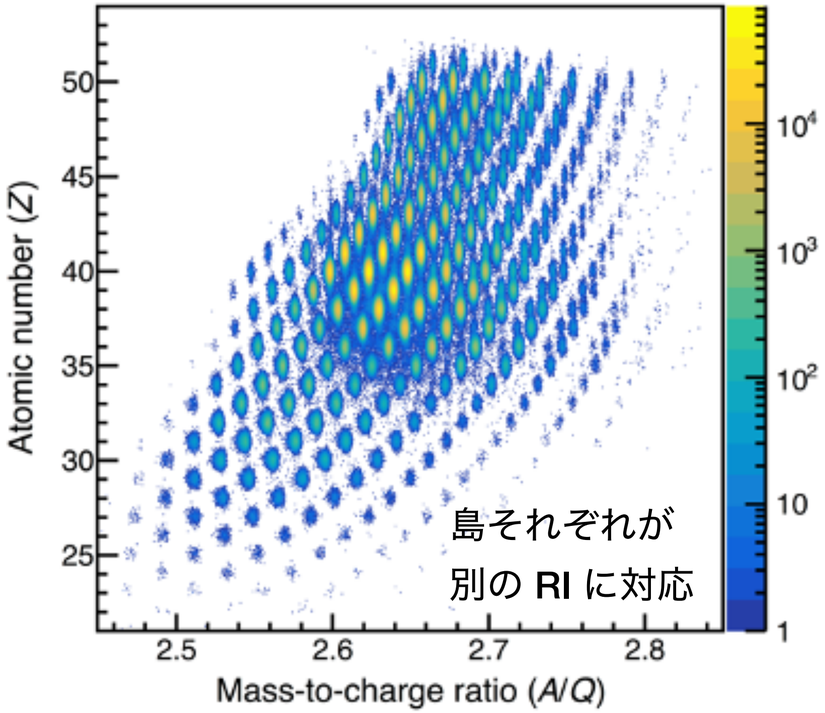
BigRIPS を用いたデータ解析の雰囲気

- F3 Plastic + F7 Plastic の TOF → β
- F3 PPAC + F5 PPAC の粒子輸送 → Bρ
- F7 IC のエネルギーロス + β → Z
- Bρ + β → A/Q

- オフライン解析で PID を用いて所望のデータを抜き出して物理の解析



PID: Particle Identification Diagram



RIBF でやりたいこと：リアルタイムデータ解析

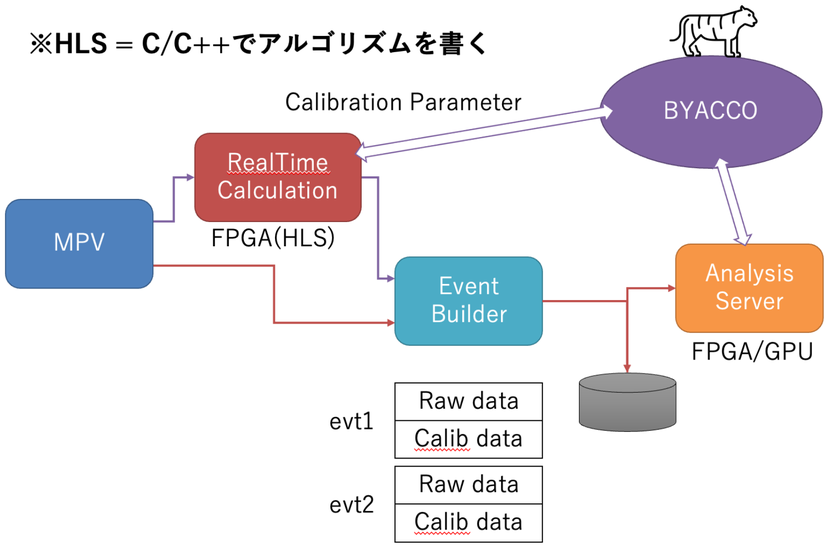
ToF・Position・ΔE といった物理量に変換済みのデータを垂れ流し

- 解析のスピードを上げる → 現状はそれぞれのグループが独自にやっている
- 物理的に意味のあるトリガーの可能性 (PID でのトリガとか？)
- 複数同時実験もやりやすくなる：それぞれの装置は施設が流している粒子識別 DAQ データを見にいくだけでよい (現在は一つの実験で粒子識別系 DAQ を占有)

ハードウェアは？

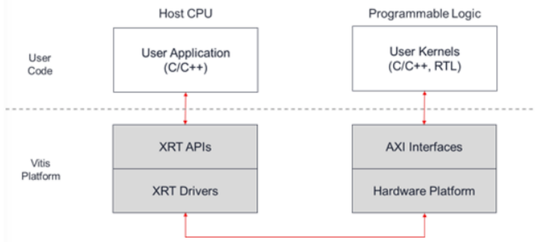
- 垂れ流されているデータの高速データ処理には FPGA などを用いたくなる
- PID のような複雑な処理を FPGA で愚直に実装するのは厳しい

→ AMD Xilinx Alveo シリーズを試してみよう



AMD Xilinx Alveo

- AMD のアクセラレータカードシリーズ
  - PCI Express 拡張カード形式でホストサーバーに FPGA の機能を追加できる
- Alveo U50
  - parallel につかえる 8GB (256MB x 32) HBM
    - 大きいデータを FPGA の近くに直接置ける
  - QSFP28 による外部との直接通信
- 統合開発環境 Vitis
  - Host CPU のアプリケーションから FPGA のカーネルを呼び出すための一連の開発フローをカバー (C++ シミュレーション・高位合成・RTL / C++ 協調シミュレーション)
  - ランタイムなどは AMD から提供される → ユーザーは基本的には FPGA をいかに活用するかを考えて C++ HLS でコードを書くのみ



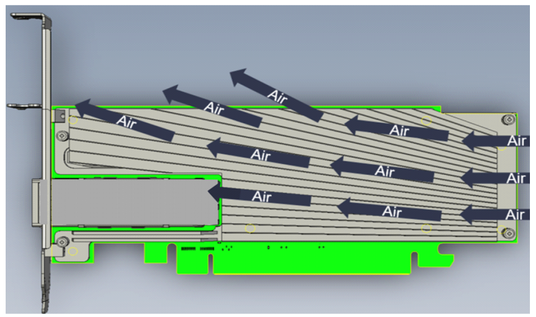
機能	ALVEO U50
アーキテクチャ	UltraScale+
フォーム ファクター	ハーフ ハイト、ハーフ レングス、シングル スロット、ロープロファイル
ルックアップ テーブル（LUT）	872,000
HBM2 メモリ	8GB
HBM2 帯域幅	316GB/s[1]
ネットワーク インターフェイス	1 x QSFP28（100GbE）*
クロック精度	IEEE 1588
PCI Express	PCIe Gen3 x 16、デュアル PCIe Gen4 x 8、CCIX
熱管理ソリューション	パッシブ
熱設計電力（TDP）	75W

Alveo を試す — 導入編

当面の目標：DAQ から出てくる raw data をリアルタイムで解析し、PID を実現する

- Alveo のセットアップ
  - ホストマシン環境にやや細かい条件あり
    - Ubuntu 22.04 だとカーネルが新すぎて運用ターゲットプラットフォームが入らなかったため 20.04 にダウングレード
    - 推奨メモリ > 80 GB (?!)
  - 謎の動作不良
    - カード取り付け OK、PCI も認識し、カードの validation などの通信もできたが、使っていると前触れもなく落ちる → 熱暴走だった：Alveo U50 はパッシブ冷却ボード (ラックマウントサーバーのようなエアフローがちゃんとしているシステム用)
    - 空冷のためのブロワーファンを取り付けてようやく正常に動作
- Tutorial を試したいが。。
  - Tutorial が必ずしも U50 向けに作られていない → Tutorial を U50 で走らせるために Tutorial を超える知識が必要なため、色々試行錯誤し、ようやく Vitis Tutorials の vector addition が実機で動いた
- 動いたはいいが。。
  - オーバーヘッドも大きく、特に速くない
  - そもそも本当にハードウェアアクセラレーションの効果があるのだろうか？

図 7: U50 パッシブ冷却カードのエアフローの方向



Alveo を試す — ハードウェアアクセラレーション編

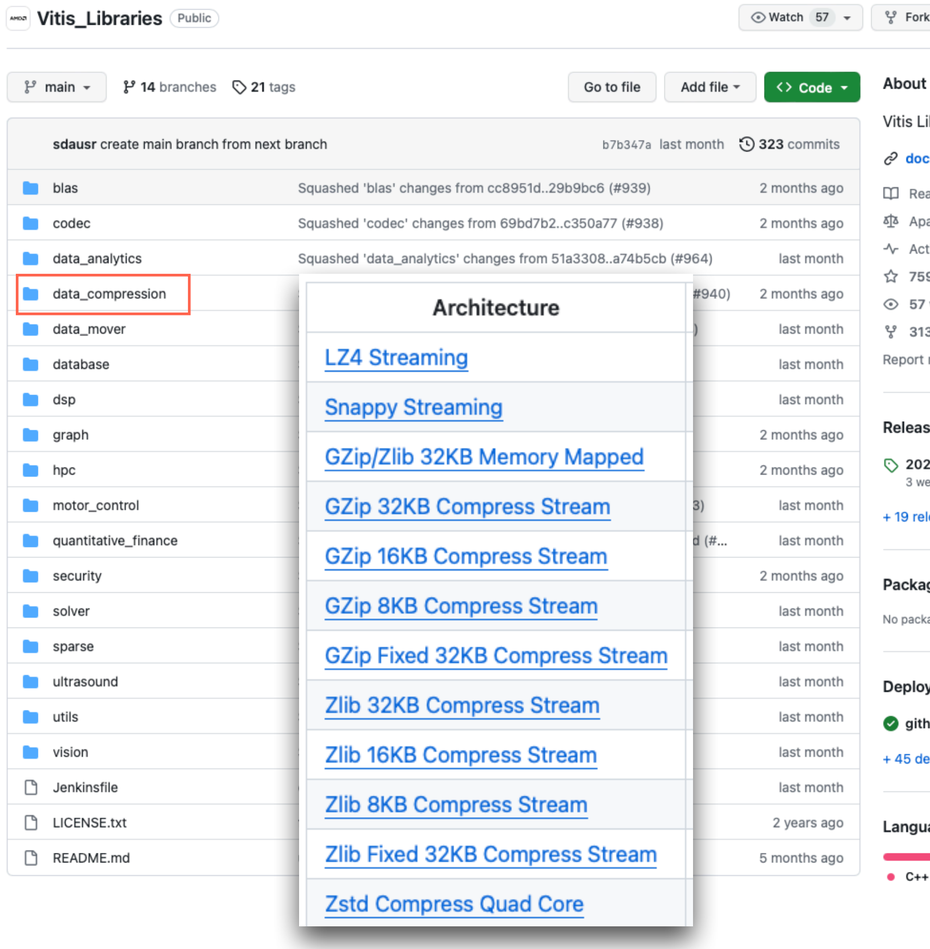
そもそもハードウェアアクセラレーション効果を感じるには、ソフトウェアコードと、equivalent な hardware-accelerated コードの両方が必要

→ Vitis data compression library の中の GZIP で比較することに

- 無数の試行錯誤
  - GitHub から落としてきた library にコードが足りない
  - ハードウェアで計算している CRC checksum の値が異なるため一見普通の gzip では解凍できないように見える

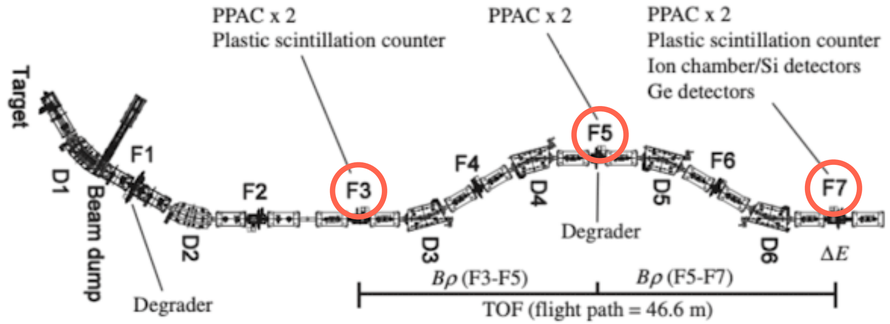
- まずまずパワフル
  - 2.5 GB ファイル圧縮で ~12 sec (CPU: ~75 sec)
    - pigz で 3.7 GHz i9 (10 core, 20 threads) だと ~6 sec
  - 解凍はそうでもない：このファイルを解凍するのに ~20 sec (CPU ~15 秒。CPU の方が速い)
  - U50 だと 8 GB HBM しかないので、少なくとも圧縮前+圧縮後の容量が 8 GB は超えられない。そのまま使うなら ~3 GB ぐらいまでか

CPU をそこそこ潤沢に使えばいいような場合に単に勝つのは難しそうだが、速いは速い → **満を持して RIBF PID へ**



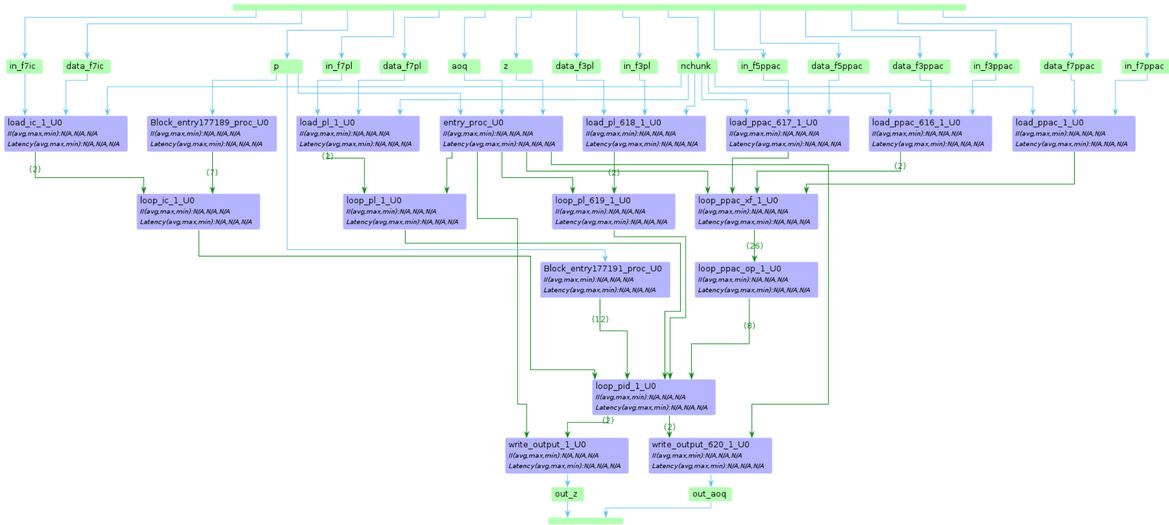
Alveo を用いた RIBF PID 1 — 問題設定

とりあえずの目標：anaroot (RIBF データ解析に標準的に使われているソフトウェアの一つ) と全く同等のプロセスによる PID を FPGA 化する



- PPAC (FP3, FP5, FP7) それぞれの位置と角度 (x, a)
  - 1 FP に 4 PPACs 存在。raw 値から反応位置を計算後、反応位置と検出器位置から最小二乗法で荷電粒子の通過位置と角度を計算
- PL (FP3, FP7) それぞれでの時間
  - 1 FP に二つのフォトマル。平均から時間を計算
- IC (FP7) でのエネルギーデポジット
  - 1 FP に 6 個の IC。それぞれの energy deposit の raw 値からペデスタルを引き、相乗平均を取り、線形変換してエネルギーを計算

- PL の情報から TOF を用いて β を、PPAC の情報と β から Bρ を計算、IC から Bethe-Bloch で Z を計算などをして、A/Q と Z を導出する



作るべきもの → **PPAC3, PPAC5, PPAC7, PL3, PL7, IC7 それぞれの raw セグメントを同時に受け取り、A/Q と Z に対応する 2 つの double 値を返すコード**

Alveo を用いた RIBF PID 2 — とりあえず試す

既存のソフトウェアコードがあればいきなりハードウェアアクセラレーションを試せるというものでもなかった

- malloc や std::vector などの動的メモリ確保は FPGA に移植できない
- ROOT の機能を使っている部分は高位合成できない (TMatrix, TMath …)
- 元のソフトが種々のオーバーヘッドのため巨大で複雑で遅い → アクセラレーションの効果をちゃんと見極められない

Step 1: ソフトウェアコードを作る

- RIDF データ (RIBF DAQ 標準データフォーマット) をデコードし、等価な結果を出力するコードを pure C++ で書き直す
  - ベンチマーク速度計測 → CPU だとレイテンシ 450 ns ぐらい (O3, 3.7 GHz i9 シングルスレッド)

Step 2: 上記コードを愚直に HLS にかけてみる

- HLS 自体は通ったが、LUT 使用率が 97% などとなった。インプリは当然上手くいかなかった。

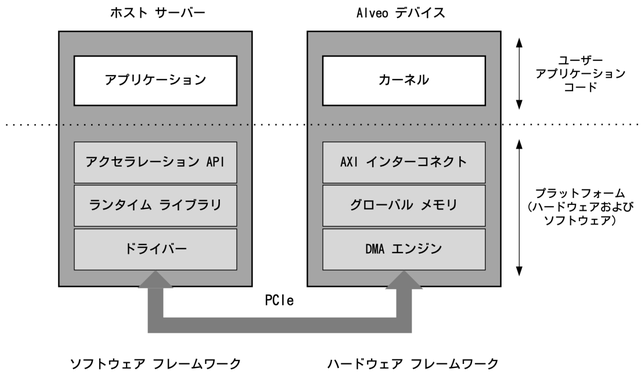
Step 3: 必要のない計算 (y, bとか) などを減らしてリソースを減らす

- HLS は通り、インプリも通った。TNS がすごい値になった。動かしてみるとデッドロック。手動解決はできなかった。

この辺までは雰囲気でやっていたが、流石にシミュレーションなどきちんとやる必要を感じ、Vitis HLS のフローに従ってやることに

Vitis を用いた開発フローについて

図 1: プラットフォームの概要

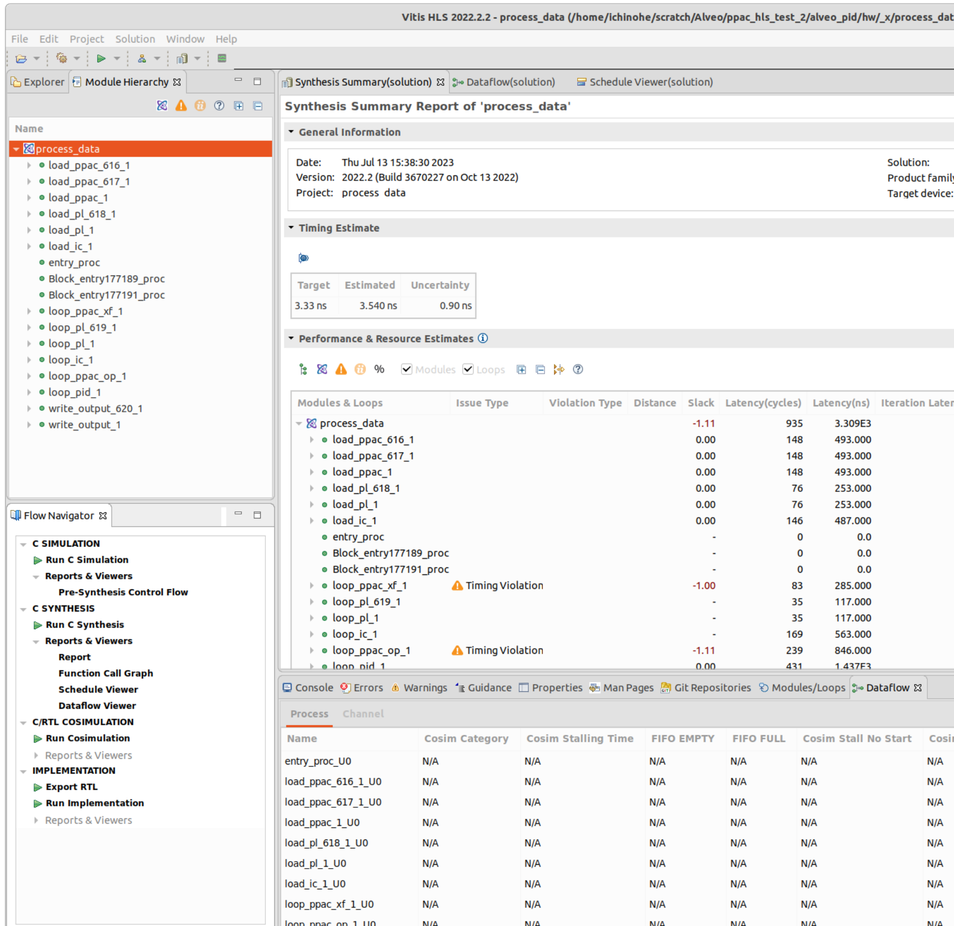


大まかな流れ

- C++ でコードを書き、vitis を用いて高位合成し、Xilinx object ファイルを作る
- Xilinx object ファイルをターゲットプラットフォーム (Alveo の FPGA や I/O などの構成) とリンクして、FPGA 用のバイナリを作成する
- FPGA バイナリを呼び出すホストコードを C++ / Python などで書く

Vitis HLS — 上から順番にやっていくといい感じにカーネルオブジェクトを作ることができる

- C simulation — C++ のままのカーネルコードと C++ のテストベンチを使い、ソフトウェアとして (アルゴリズム的に) 正しく動作するか調べる
- C synthesis — カーネルコードから HLS で RTL を生成し、リソースの見積もりやパイプライン化できているかのチェックなどを行う
- C/RTL cosimulation — HLS で生成された RTL と C++ のテストベンチを使い、ホストと FPGA の連携や、FPGA の中のデッドロック検知などを行う。
- Implementation — Vivado でのリンクなどに用いることができるカーネルオブジェクトを生成する



Alveo を用いた RIBF PID 3 — Vitis HLS フロー

Step 4: 元のコードをベースに C++ テストベンチを作成し、前の方から問題を全て潰していく作戦に変更

- HLS 合成で出てくる II (iteration interval) や timing violation の問題を潰すようにコードを大幅に変更
  - pragma — HLS PIPELINE (指定したループをパイプライン化), HLS ALLOCATION (指定した関数や演算のリソースを制限, HLS UNROLL (指定したループを展開し並列化), HLS INLINE (指定した関数をインライン化) etc.
  - dataflow 化 — それぞれの細かいタスクに分け、間をパイプラインレジスタで繋ぐ (処理が分離するのでタスク並列できるようになる)

Step 5: C/RTL 協調シミュレーション

- C のテストベンチと HLS で作った RTL を組み合わせて行うシミュレーション
  - cosimulation を走らせること自体にも結構難儀 (pragma のパラメータを適切な値に設定しないと動かない)
- デッドロックがこの段階で検出でき、データのロードに時間がかかっていることが原因であることが判明

```
                    const int nchunk) {

  loop_ppac_op: for (int i = 0; i < nchunk; ++i) {
#pragma HLS PIPELINE
#pragma HLS LOOP_TRIPCOUNT min=MIN max=MAX
#pragma HLS ALLOCATION function instances=compute_x_a limit=1

      double _f3_x_1a = f3_x_1a.read();
      double _f3_x_1b = f3_x_1b.read();
      double _f3_x_2a = f3_x_2a.read();
      double _f3_x_2b = f3_x_2b.read();
      double _f3_f_1a = f3_f_1a.read();
      double _f3_f_1b = f3_f_1b.read();
      double _f3_f_2a = f3_f_2a.read();
      double _f3_f_2b = f3_f_2b.read();
      compute_x_a(_f3_x_1a, _f3_f_1a, _f3_x_1b, _f3_f_1b, _f3_x_2a, _f3_f_2a, _f3_x_2b, _
                  f3_opx, f3_opa,
                  p_f3_1a, p_f3_1b, p_f3_2a, p_f3_2b);

      double _f5_x_1a = f5_x_1a.read();
```

```
#pragma HLS STREAM variable=f7t     type=fifo depth=3
#pragma HLS STREAM variable=f7s     type=fifo depth=3

#pragma HLS STREAM variable=_aoq    type=fifo depth=2
#pragma HLS STREAM variable=_z      type=fifo depth=2

#pragma HLS DATAFLOW

    const ppac_params p_f3ppac_1a = p.f3ppac_1a;
    const ppac_params p_f3ppac_1b = p.f3ppac_1b;
    const ppac_params p_f3ppac_2a = p.f3ppac_2a;
    const ppac_params p_f3ppac_2b = p.f3ppac_2b;
    const ppac_params p_f5ppac_1a = p.f5ppac_1a;
    const ppac_params p_f5ppac_1b = p.f5ppac_1b;
    const ppac_params p_f5ppac_2a = p.f5ppac_2a;
    const ppac_params p_f5ppac_2b = p.f5ppac_2b;
    const ppac_params p_f7ppac_1a = p.f7ppac_1a;
    const ppac_params p_f7ppac_1b = p.f7ppac_1b;
    const ppac_params p_f7ppac_2a = p.f7ppac_2a;
    const ppac_params p_f7ppac_2b = p.f7ppac_2b;
    const pl_params   p_f3pl      = p.f3pl;
    const pl_params   p_f7pl      = p.f7pl;
    const ic_params   p_f7ic      = p.f7ic;
    const pid_params  p_pid       = p.pid;


    load_ppac(data_f3ppac, f3ppac, nchunk);
    load_ppac(data_f5ppac, f5ppac, nchunk);
    load_ppac(data_f7ppac, f7ppac, nchunk);
    load_pl(data_f3pl, f3pl, nchunk);
    load_pl(data_f7pl, f7pl, nchunk);
    load_ic(data_f7ic, f7ic, nchunk);

    loop_ppac_xf(f3ppac, f5ppac, f7ppac,
                 f3_x_1a, f3_f_1a, f3_x_1b, f3_f_1b, f3_x_2a, f3_f_2a, f3_x_2b, f3_f_2b,
                 f5_x_1a, f5_f_1a, f5_x_1b, f5_f_1b, f5_x_2a, f5_f_2a, f5_x_2b, f5_f_2b,
```

Alveo を用いた RIBF PID 4 — やっと動いた

Step 6: デッドロックが起きないようにコードの変更

- 全ての検出器のデータを一気に読み込み処理していたが、それぞれの検出器データを別の HBM port から並列に読むように変更 (テストベンチから書き直しになる)。

Step 7: 全ての問題を解決し、インプリを試すが、失敗

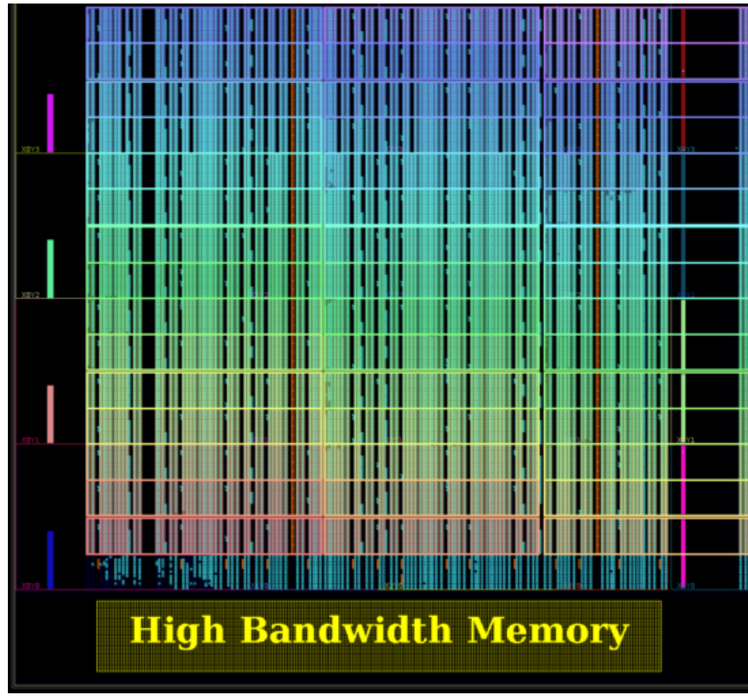
- congestion level が高すぎ (配線混雑レベルが高すぎて上手くインプリできない)

Step 8: インプリを通そうと試行錯誤

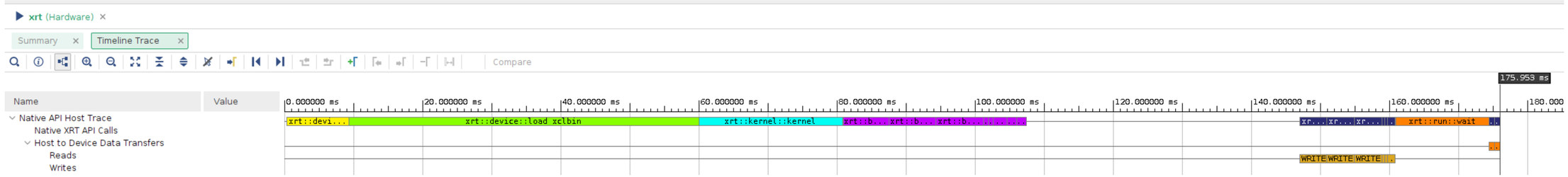
- 使う HBM の物理的場所を分散する、並列処理している部分をパイプライン処理にする (リソースは減るが、II (Iteration Interval) が増えてしまう。トレードオフ) など、pragma などで調整し、さらにリソースを減らす
- negative slack は残っていたが、cosim でデッドロックは起きないこともわかったため、インプリさえうまくいけば多少遅くても動くとは思いインプリを試すことに → 成功

Step 9: やっと動いた

- **ターゲット周波数 300 MHz に対し、220.4 MHz が実現。実際にデッドロックなく動作した**



パフォーマンス



- 実現したクロック：1 セット処理するために必要なクロック：~1000 @ 220 MHz → レイテンシ ~4.3 us
- 実現したパイプライン処理：5 クロックごと → スループット 570000 evs / 13 ms
  - CPU では 450 ns なので、シリアル処理にすると 570000 evs / 250 ms ぐらい
  - **20 倍のスループットが出ている。20 並列相当。速い。**

- 5 clk でパイプライン処理できるため、40 MHz ぐらいでデータを受け付けられるということ。
- 5 us ぐらい待てるなら、例えば PID 情報を使ってトリガーをかけるなんてことも可能。

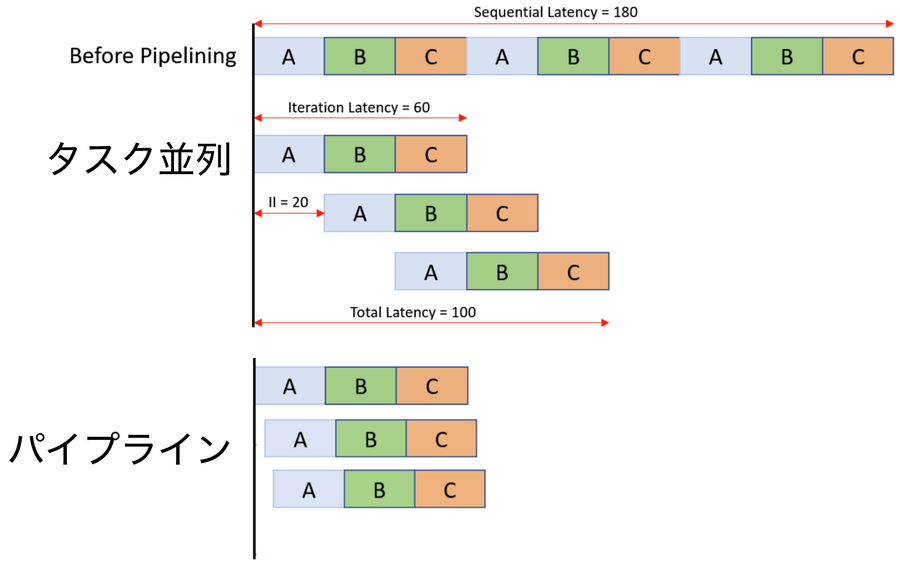
実際に動いての反省点

- リソースが多すぎるとインプリは通らない。
- デッドロックとかは cosim でわかるはずなので、ちゃんと Vitis HLS フローに従って開発をした方が結果的に効率がいい
- 何も考えずに C++ を書き、HLS をすればうまくいくような気に最初はなるが (現にコードが小さい時はうまくいく)、コードが複雑になってくるとハードウェアの気持ちになった書き方をする必要が結局出てくる
  - 例：タスク並列化した時、II に極端に差があるとデッドロックが発生するのでそれを揃えるようなデータの読み方にする

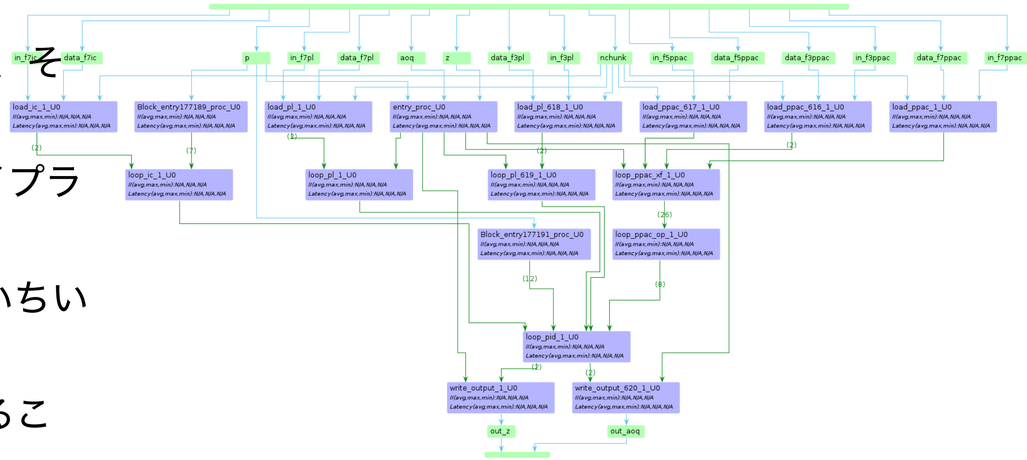
使い所に関する所見

データ処理の高速化の方法

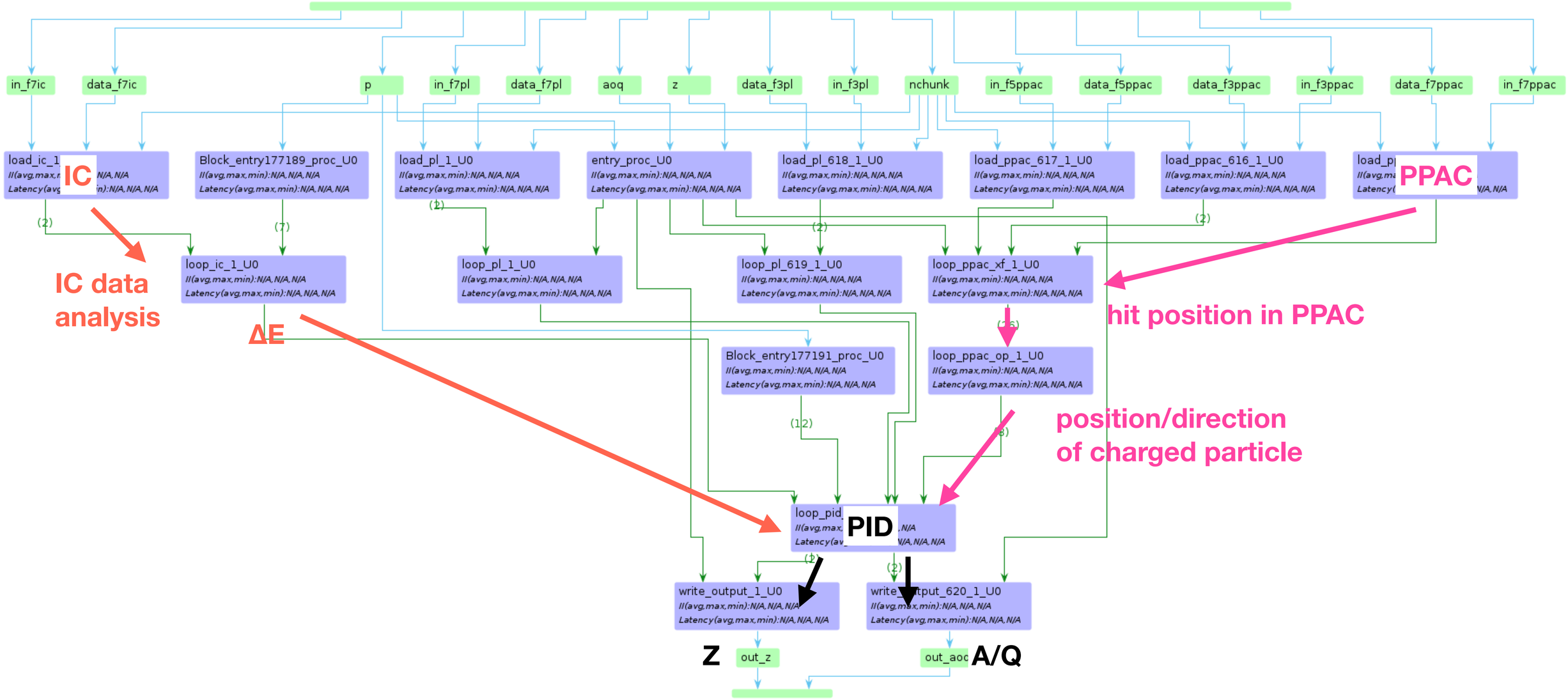
- 処理速度そのものを上げる (クロックを速くする)。CPU の方針
- 並列化する
  - データ並列 (GPU; 1000 とか並列にできる) — 同じ処理をたくさんのデータにやる。処理自体は簡単だが、データがものすごく多い (毎フレームすべてのピクセル 1920x1080 に関して二値化処理するとか) 時など



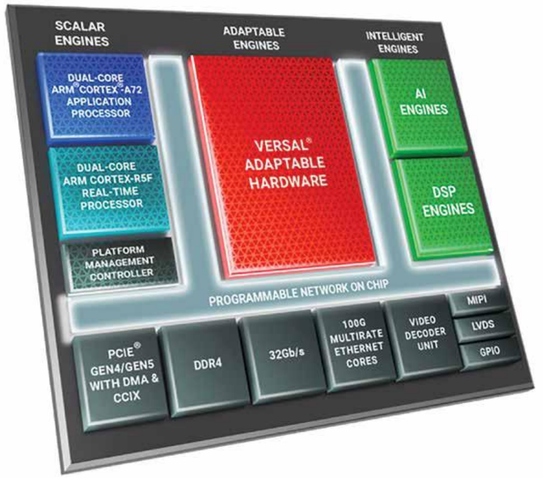
  - **タスク並列 (FPGA が向いていそう)**
    - 色々な独立にできる処理がある時は、うまく処理を分けるとパイプライン化できる。複数タイプの入力に、並列に別の処理をして、後段でマージなど、複雑なことができる
    - HLS を使えば比較的複雑なロジックを専用のハードウェアにできるため、それぞれもそこそこ速い。内部の一部を並列化などもできる
    - 全体のレイテンシは CPU になかなか敵わないが、数クロックとかでパイプライン処理できるためスループットを大幅に稼げる
    - パイプラインバッファで流れていくため、レジスタへの読み書きなどがいちいち行われない
    - QSFP28 などの I/O を用いて、外部からストリーミングでデータを受けることもできる (普通の GPU にはない特徴)



Full data flow of PID procedure

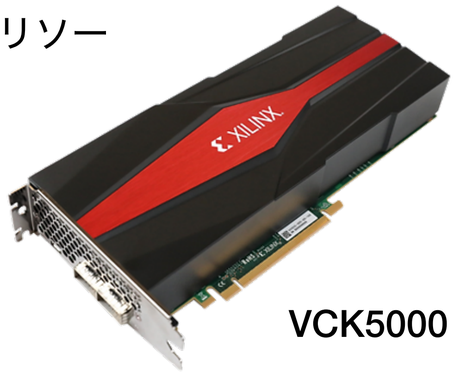


今やっていること・展望



アクセラレータカードの可能性を探り中

- 外部 I/O を用いた通信
  - Xilinx Aurora 64B/66B を用いた 100 Gbps 通信 → ループバックだけはできた (Tutorial の改造は必要)
  - 確かに 100 Gbps 出ている。Versal を導入したら実際にデータ通信を試す予定
- ドリフトチェンバーのデータ解析 — 現状のシンプルなアルゴリズムの高速化ができないか？ → 難航中
  - ヒット数可変。あり得るすべてのパターンのうちベストなものを取り出す
    - ハードウェアになってしまう以上可変長の構造 (データ・ループ) に弱い
  - ある N ヒットパターンに対して、ワイヤーに対する “左右” の自由度があるため、2^N の可変長ループが内側で回る
    - 多重ループ：パイプライン処理するには内側を II 以内に終える必要あり。並列にするとリソース消費が膨大
  - 前もってループ回数を計算してループ最内タスクのみパイプラインで回す → ループ最内タスクだけの時間で見れば確かに速くなるが、パラメータ生成と後処理・FPGA へのデータ転送のオーバーヘッドが大きく得しない
  - 適切なパラメータを順次生成するコード・後処理のコードも FPGA 内に実装する？ → 計算上はいけそうだがリソースが足りるか？
  - 根本的にアルゴリズムを考え直した方がいいような気がする。知見がある人意見ください。



VCK5000

- Versal VCK 5000 導入
  - FPGA に加え AI コア (GPU 的なもの) がある。QSFP28 x 2。データを垂れ流せる GPU のように使えるか？
  - 開発に GPU が strongly recommended (?!)
  - ドリフトチェンバー解析を機械学習とかでやるなど試す？

まとめ

- RIBF データ処理の高度化に向けてハードウェアアクセラレーターの導入を試行錯誤している
  - PID のようなタスク並列可能な処理には結構向いていそう
  - DC データ処理のように、あまり向かなそうなタスクがあることもわかってきた
- 引き続き Aurora 通信・Versal の導入などいろいろ探っていく予定
- この辺の知見がある人、ぜひ議論させてください & 一緒にやりましょう