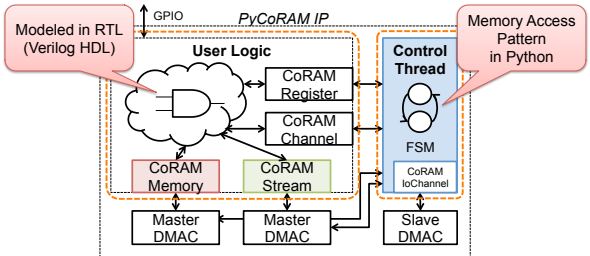
2014年10月22日 15:00-15:50
組み込みシステムシンポジウム2014 (ESS2014)
チュートリアル

PyCoRAMによる Pythonを用いたポータブルな FPGAアクセラレータ開発

高前田（山崎） 伸也
奈良先端科学技術大学院大学 情報科学研究科
E-mail: shinya_at_is_naist_jp

今日の話: PyCoRAMについて

- 高位合成技術とメモリシステム抽象化を用いた アクセラレータIPコア開発のためのフレームワーク
  - キーワード: FPGAアクセラレータ・IPコア設計・Python-Verilog 高位合成・メモリシステム抽象化・AXI4インターコネクト

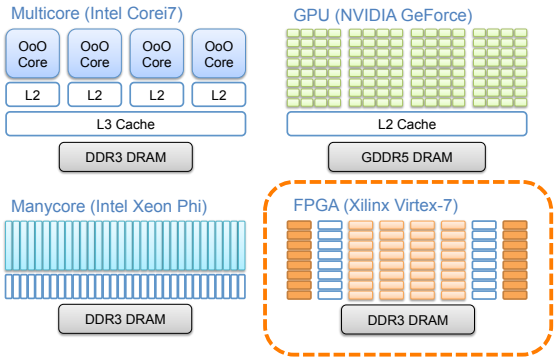


PyCoRAM and Pyverilog are released for public!

- PyCoRAM
  - http://shtaxxx.github.io/PyCoRAM/
- Pyverilog
  - http://shtaxxx.github.io/Pyverilog/

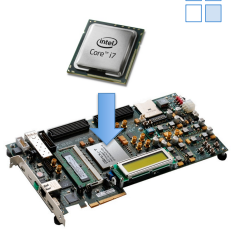
はじめに

Heterogeneous Computing

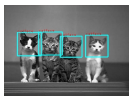


FPGAs in Anywhere

- As an LSI verification platform
  - Testing the LSI functions, before the actual fabrication
    - It enables longer behavior simulations than SW-based simulators



- As a final product
  - Vision processing
  - Network router
  - Broadcasting
  - Stock Trading (High Frequency Trading (HFT))
  - Oil mining

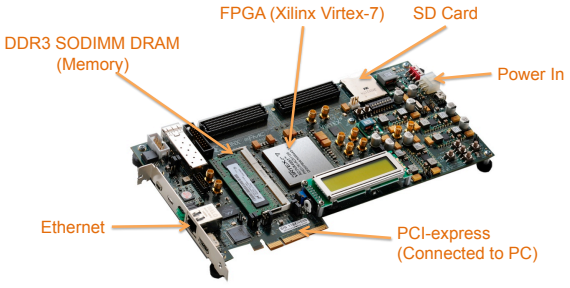


Object Detection

Stock Trading

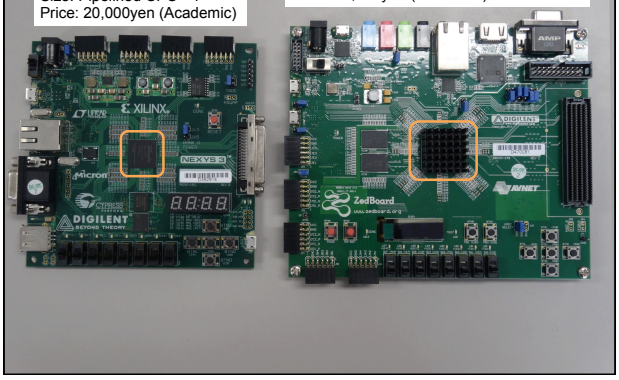
FPGA Board Overview

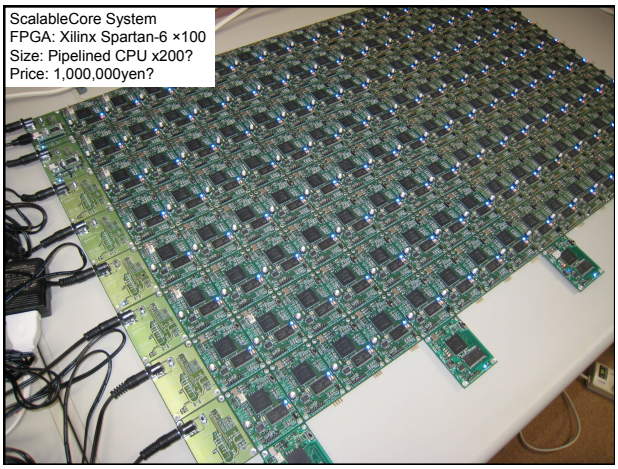
Xilinx VC707 Evaluation Board (400,000yen)



Digilent Atlys
FPGA: Xilinx Spartan-6 LX45
Size: Pipelined CPU ×4
Price: 20,000yen (Academic)

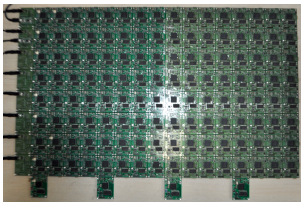
Digilent ZedBoard
FPGA: Xilinx Zynq-7020
Size: Pipelined CPU ×8 (+ ARM DualCore)
Price: 50,000yen (Academic)





アプリケーションのポータビリティ

- どうやって様々なプラットフォームをサポートするか?
  - プラットフォーム毎に特性が違う☹
    - ロジックサイズ・メモリサイズ・メモリインターフェース・I/O・・・



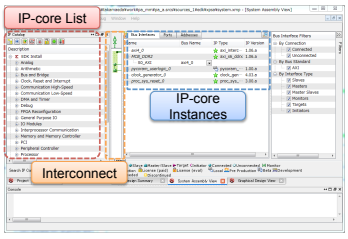
ScalableCore System (our FPGA system)
(Xilinx Spartan-6 LX16 × 128-node)

Digilent Atlys
(Xilinx Spartan-6 LX45)

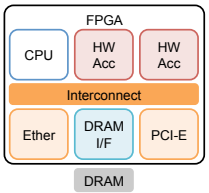
Xilinx ML605
(Xilinx Virtex-6 LX240T)

IPコアベースのシステム開発

- IPコアを開発・追加して繋げばシステム完成☺
  - 標準的なインターコネクトでIPコア達を接続
  - EDAツールが自動的にインターコネクトと（いくつかの）デバイス依存のインターフェースを生成してくれるから楽ちん



Xilinx Platform Studio (XPS)



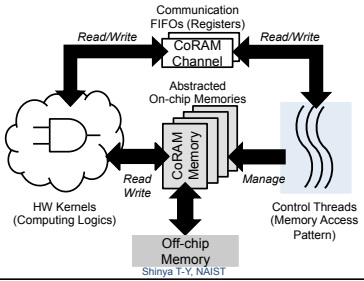
どうやってアクセラレータIPを実装するか？

- 普通にHDLでアクセラレータを実装するのは芸が無い
  - というかいろいろ面倒で嫌だ！
    - 演算とメモリアクセスのスケジューリングロジック
      - ダブルバッファリングとか面倒
    - メモリシステムの制御回路
      - HDLでステートマシンを書くのは面倒だし間違えやすい
    - デバッグが面倒
  - でもパイプラインの振る舞いはサイクルレベルで定義したい
    - FPGAで性能を出すには高稼働率のパイプラインが重要
      - だから計算ロジックはHDLで書きたい
      - 高位合成だとチューンがイマイチ難しい
- 抽象化されたメモリシステムが使えると幸せそう

CoRAMメモリアーキテクチャ

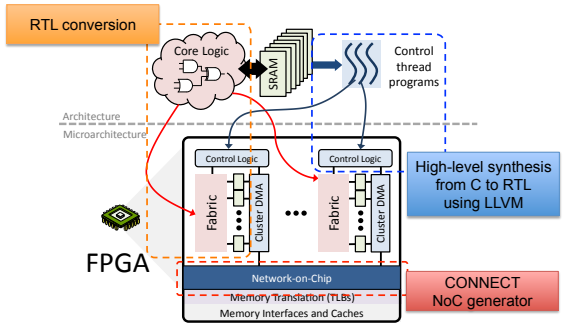
CoRAM [Chung+,FPGA'11]

- FPGAアクセラレータのためのメモリ抽象化
  - 高位モデルによるメモリ管理でアクセラレータをポータブルに
    - 計算カーネルとメモリアクセスの分離
    - ソフトウェアのモデルによるメモリアクセスパターンの記述



What "Runs" CoRAM?

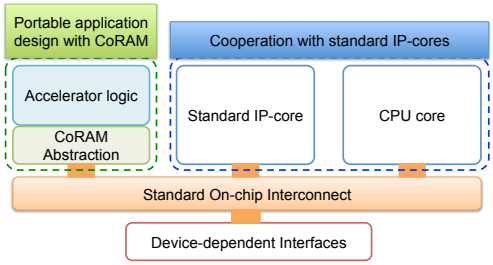
From CoRAM Tutorial @FPGA'13



PyCoRAM

Motivation: CoRAM for Modern EDKs

- CoRAMのメモリ抽象化を今時のEDKで使いたい
  - 標準的なインターコネクト（≒AMBA AXI4）に繋ぎたい
  - そうすれば他の普通のIPコアとも簡単に共存できそう



PyCoRAM [Takamaeda+,CARL'13]

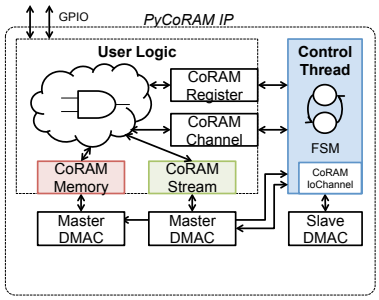
- ベンダーEDK向けのPythonベースのCoRAM実装
  - 計算カーネルのRTL記述とメモリアクセスパターンの Python記述からAXI4 IPコアを自動合成
  - 出来上がったIPコアをEDKでポチポチつなげばシステム完成！
- 特徴
  - Pythonでのコントロールスレッド記述
    - Pythonで簡単にメモリアクセスパターンを記述できる
      - 独自の高位合成コンパイラでPython記述からVerilog HDLのRTLを合成
  - AMBA AXI4インターコネクトのサポート
    - Xilinx Platform Studio (XPS)などを用いたIPコアベースの開発を支援
  - 計算ロジックの複雑なデザインに対応
    - ハードウェアデザイン解析・生成のための オープンソースツールキットPyverilogを活用

オリジナルのCoRAMとの比較

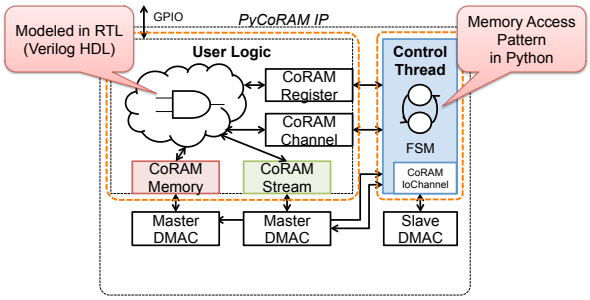
	CoRAM	PyCoRAM
Language for Control-Thread	C	Python
Supported Memory Operations	(Blocking/Non-Blocking) Read/Write	(Blocking/Non-Blocking) Read/Write
On-chip Interconnect	CONNECT NoC [FPGA'12]	AMBA AXI4
FSM Granularity in Control Thread	LLVM-IR	Python AST Node
Generate Statement Support for User logics	No	Yes
Supported FPGAs	Xilinx ML605 Altera Terasic DE-4	Any FPGAs supporting AXI Bus
# Lines of Code	11,682 lines (w/o CONNECT)	4,947 lines (w/o Pyverilog)

FSM: Finite State Machine
LLVM-IR: Low Level Virtual Machine Intermediate Representation
AST: Abstract Syntax Tree

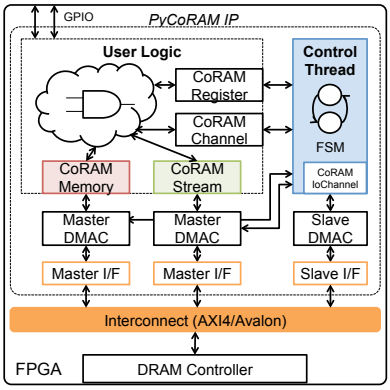
PyCoRAMマイクロアーキテクチャ



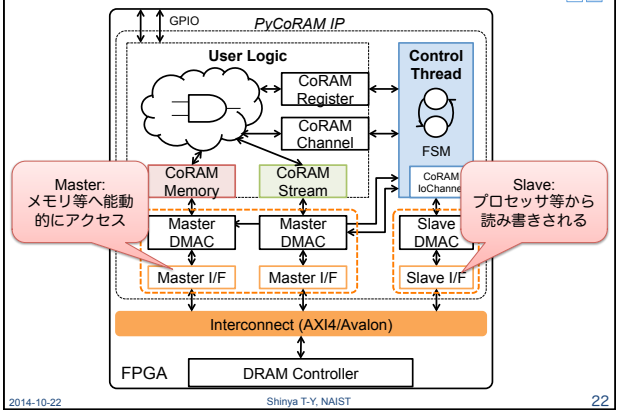
PyCoRAMマイクロアーキテクチャ



PyCoRAMマイクロアーキテクチャの実装

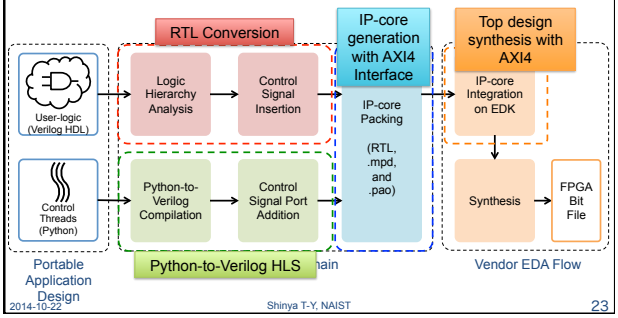


PyCoRAMマイクロアーキテクチャの実装



PyCoRAMにおける開発フロー

- 計算カーネルのRTLとPythonでのコントロールスレッド記述からIPコアパッケージを生成
  - 生成されたIPコアは通常のEDAフローで利用可能



PyCoRAMの使い方

PyCoRAMにおけるIPコアの作り方・でき方

- 2種類のファイルを用意する
  - Verilog HDL: 計算ロジック
  - Python: コントロールスレッド（メモリアクセスパターン）

```
CoramMemory1P
#(
 .CORAM_THREAD_NAME("thread_name"),
 .CORAM_ID(0),
 .CORAM_ADDR_LEN(ADDR_LEN),
 .CORAM_DATA_WIDTH(DATA_WIDTH)
 )
inst_memory
(.CLK(CLK),
.ADDR(mem_addr),
.D(mem_d),
.WE(mem_we),
.Q(mem_q)
);
```

```
def calc_sum(times):
    ram = CoramMemory(idx=0, datawidth=32, size=1024)
    channel = CoramChannel(idx=0, datawidth=32)
    addr = 0
    sum = 0
    for i in range(times):
        ram.write(0, addr, 128)
        channel.write(addr)
        sum += channel.read()
        addr += 128 * (32/8)
    print('sum=', sum)
calc_sum(8)
```

- PyCoRAMが自動的にIPコアのパッケージを作成
  - Python-Verilog高位合成とRTL変換を自動で行う

計算ロジックにおけるCoRAMオブジェクト

- CoRAMオブジェクトはブロックRAMやFIFOとして扱う
  - 一般的なメモリとよく似たインターフェース
    - CoRAMと外部を接続するインターフェースは自動的に追加される
  - いくつかのパラメータで特性を指定
    - スレッド名，ID，データ幅，アドレス幅，スキャッターギャザー等

```
CoramMemory1P
#(
 .CORAM_THREAD_NAME("thread_name"),
 .CORAM_ID(0),
 .CORAM_ADDR_LEN(ADDR_LEN),
 .CORAM_DATA_WIDTH(DATA_WIDTH)
 )
inst_memory
(.CLK(CLK),
.ADDR(mem_addr),
.D(mem_d),
.WE(mem_we),
.Q(mem_q)
);
```

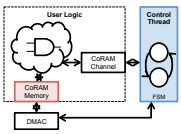
(a) CoRAM Memory

```
CoramChannel
#(
 .CORAM_THREAD_NAME("thread_name"),
 .CORAM_ID(0),
 .CORAM_ADDR_LEN(CHANNEL_ADDR_LEN),
 .CORAM_DATA_WIDTH(CHANNEL_DATA_WIDTH)
 )
inst_channel
(.CLK(CLK),
 .RST(RST),
 .D(comm_d),
 .ENQ(comm_enq),
 .FULL(comm_full),
 .Q(comm_q),
 .DEQ(comm_deq),
 .EMPTY(comm_empty)
 );
```

(b) CoRAM Channel

Pythonによるコントロールスレッド

- CoRAMオブジェクトに対する処理を記述する
  - CoRAMメモリ（read/write）
    - オンチップCoRAMメモリとオフチップDRAMとの間のDMA転送によるデータ移動
  - CoRAMチャネル（read/write）
    - ユーザロジックとコントロールスレッドとの間のトークンのやりとり



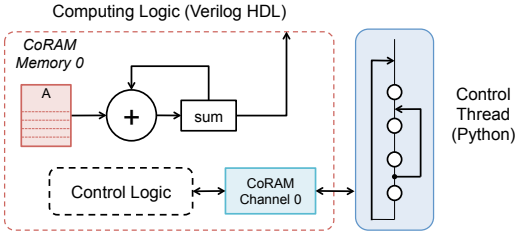

```
0  def calc_sum(times):
1      ram = CoramMemory(idx=0, datawidth=32, size=1024)
2      channel = CoramChannel(idx=0, datawidth=32)
3      addr = 0
4      sum = 0
5      for i in range(times):
6          ram.write(0, addr, 128)  # Transfer (off-chip DRAM to BRAM)
7          channel.write(addr)      # Notification to User-logic
8          sum += channel.read()    # Wait for Notification from User-logic
9          addr += 128 * (32/8)
10     print('sum=', sum)           # $display Verilog system task
11 calc_sum(8)
```


サポートされているPyCoRAMオブジェクト

- データ置き場（メモリ・ストリーム）
  - CoramMemory
    - Block RAM that the data is replaced by the control thread
  - CoramInStream
    - Input FIFO from off-chip DRAM
  - CoramOutStream
    - Output FIFO to off-chip DRAM
- ユーザロジックとコンロールスレッド間のチャネル
  - CoramChannel
    - FIFO between user-logic and control-thread
  - CoramRegister
    - Latch between user-logic and control-thread
- 他のIPコア・プロセッサからアクセスできるスレーブチャネル
  - CoramIoChannel
    - (AXI4/Avalon) Slave interface

例: 配列の和を求めるアクセラレータ

- 1-CoRAM+1-Threadの簡単なハードウェア
  - CoRAMメモリにDRAM（など）からデータを読み込む
  - コントロールスレッドでオンチップ-オフチップ間のデータ転送パターンを表現



計算ロジック (1): I/Oポート

```
`include "pycoram.v"

`define THREAD_NAME "ctrl_thread"

module userlogic #
  (
   parameter W_A = 10,
   parameter W_COMM_A = 4,
   parameter W_D = 32,
   parameter SIZE = 128
   )
  (
   input CLK,
   input RST,
   output reg [31:0] sum
   );

  reg [W_A-1:0] mem_addr;
  reg [W_D-1:0] mem_d;
  reg           mem_we;
  wire [W_D-1:0] mem_q;

  reg [W_D-1:0]  comm_d;
  reg            comm_enq;
  wire           comm_full;
  wire [W_D-1:0] comm_q;
  reg            comm_deq;
  wire           comm_empty;

  reg [3:0] state;
```

クロック(CLK)とリセット(RST)以外に専用のI/Oは不要

CoRAMメモリのための信号(BRAMと同じインターフェース)

CoRAMチャネルのための信号(FIFOと同じインターフェース)

ステートマシン用変数

計算ロジック (2): パイプライン/FSM

```
always @(posedge CLK) begin
  if(RST) begin
    state <= 0;
    mem_we <= 0;
    comm_deq <= 0;
    comm_enq <= 0;
  end else begin
    // default value
    comm_enq <= 0;
    comm_deq <= 0;

    if(state == 0) begin
      sum <= 0;
      mem_d <= 0;
      mem_we <= 0;
      mem_addr <= 0;
      if(!comm_empty) begin
        comm_deq <= 1;
        state <= 1;
      end
    end else if(state == 1) begin
      state <= 2;
      mem_addr <= 0;
    end else if(state == 2) begin
      state <= 3;
      mem_addr <= mem_addr + 1;
```

```
    end else if(state == 3) begin
      mem_addr <= mem_addr + 1;
      sum <= sum + mem_q;
      if(mem_addr == SIZE-2) begin
        state <= 4;
      end
    end else if(state == 4) begin
      state <= 5;
      sum <= sum + mem_q;
    end else if(state == 5) begin
      state <= 6;
      sum <= sum + mem_q;
    end else if(state == 6) begin
      if(!comm_full) begin
        comm_d <= sum;
        comm_enq <= 1;
        state <= 7;
      end
    end else if(state == 7) begin
      comm_enq <= 0;
      if(!comm_empty) begin
        comm_deq <= 1;
        state <= 1;
      end
    end
  end
```

CoRAMチャネルから読み出し（コントロールスレッドから受信）

CoRAMチャネルに書き込み（コントロールスレッドに通知）

計算ロジック (3): 子インスタンス

```
CoramMemory1P
#(
  .CORAM_THREAD_NAME(`THREAD_NAME),
  .CORAM_ID(0),
  .CORAM_SUB_ID(0),
  .CORAM_ADDR_LEN(W_A),
  .CORAM_DATA_WIDTH(W_D)
  )
inst_data_memory
(.CLK(CLK),
 .ADDR(mem_addr),
 .D(mem_d),
 .WE(mem_we),
 .Q(mem_q)
 );

CoramChannel
#(
  .CORAM_THREAD_NAME(`THREAD_NAME),
  .CORAM_ID(0),
  .CORAM_ADDR_LEN(W_COMM_A),
  .CORAM_DATA_WIDTH(W_D)
  )
inst_comm_channel
(.CLK(CLK),
 .RST(RST),
 .D(comm_d),
 .ENQ(comm_enq),
 .FULL(comm_full),
 .Q(comm_q),
 .DEQ(comm_deq),
 .EMPTY(comm_empty)
 );

endmodule
```

CoRAMメモリ(BRAMと同じインターフェース)

CoRAMチャネル(FIFOと同じインターフェース)

コントロールスレッド (Python)

ram (CoRAMメモリ)とchannel (CoRAMチャネル)の宣言

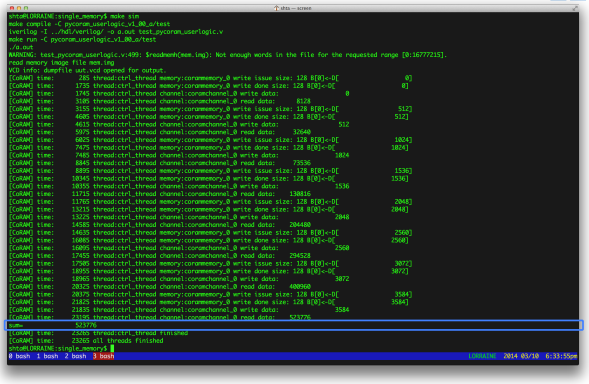
```
def ctrl_thread():
    ram = CoramMemory(idx=0, datawidth=32, size=1024, length=1, scattergather=False)
    channel = CoramChannel(idx=0, datawidth=32, size=16)
    addr = 0
    sum = 0
    for i in range(8):
        ram.write(0, addr, 128) # from DRAM to BlockRAM
        channel.write(addr)
        sum = channel.read()
        addr += 512
    print('sum=', sum)

ctrl_thread()
```

CoRAMメモリにDMA転送したり
CoRAMチャネルから読んだり書いたり

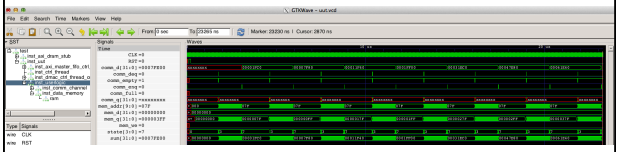
コンパイル

```
shta@LORRAINE:single_memory$ make build
python3.3 -m ../../../pycoram.py -I ../../../include/ -t userlogic --ipver=v1_00_a --extaddrwidth=32 --extdatawidth=
512 --signalwidth=64 --memimg=../mem-incr.hex  --simaddrwidth=24 --hperiod_ulogic=5 --hperiod_cthread=5 --hperiod_ax
i=5 --singleclock  userlogic.v ctrl_thread.py
Generating LALR tables
WARNING: 146 shift/reduce conflicts
Generating LALR tables
WARNING: 146 shift/reduce conflicts
CoRAM Objects in User-defined RTL
  CoRAM CoramMemory
    CoramMemory(ID:0[0] Thread:ctrl_thread AddrWidth:(9, 0) DataWidth:(31, 0))
  CoRAM CoramChannel
    CoramChannel(ID:0 Thread:ctrl_thread AddrWidth:(3, 0) DataWidth:(31, 0))
CoRAM Objects in Control-Thread 'ctrl_thread', # FSM = 15
  CoRAM CoramMemory:
    CoramMemory(ID:0 Length:1 AddrWidth:10 DataWidth:32) alias: __s0_ram
  CoRAM CoramChannel:
    CoramChannel(ID:0 AddrWidth:4 DataWidth:32) alias:  __s0_channel
shta@LORRAINE:single_memory$ ls pycoram_userlogic_v1_00_a/
data doc  hdl  test
shta@LORRAINE:single_memory$
```


シミュレーション結果



シミュレーション結果（波形）



評価

評価

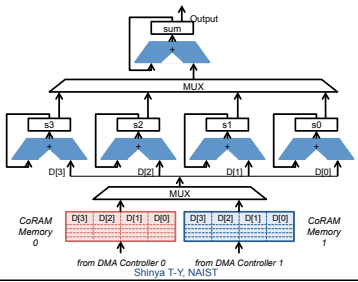
- 評価項目: メモリバンド幅利用率
  - メモリ抽象化の下でも高いメモリ性能を達成できることを示す
- セットアップ
  - FPGAボード2種
    - Digilent Atlys
      - Spartan-6 LX45
      - DDR2-800 DRAM 128MB (1.2GB/s*)
        *300MHz動作のため
      - AXI4 128-bit, 100MHz (1.6GB/s)
    - Xilinx ML605
      - Virtex-6 LX240T
      - DDR3-800 DRAM 512MB (6.4GB/s)
      - AXI4 256-bit, 200MHz (6.4GB/s)
  - EDK
    - Xilinx Platform Studio (14.6)

Digilent Atlys
(Xilinx Spartan-6 LX45)

Xilinx ML605
(Xilinx Virtex-6 LX240T)

評価用アプリケーション

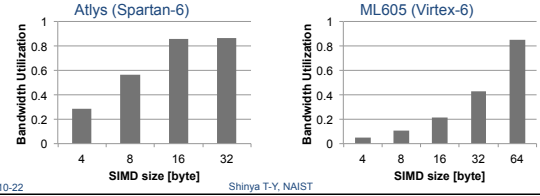
- 配列の和を求める
  - 2つのCoRAMメモリをダブルバッファで利用
  - SIMD幅 (=同時に計算する要素数) を変化させてその影響を観測
    - 4, 8, 16, 32, 64 (bytes)



結果：メモリバンド幅利用率

- 結構良い感じでメモリバンド幅を使えている
  - Atlys: ~~85.5%~~ (@ 16-byte)
  - ML605: ~~84.9%~~ (@ 64-byte)
- 100%使い切れていない理由
  - ~~各DMACのトランザクションがシーケンシャルになっているから~~
    - ~~結果，メモリアクセスレイテンシが性能にダイレクトに影響~~

大幅に改善！
（現在評価中）



応用その１
行列積・ステンシル
アクセラレータ

高性能コンピュータシステム設計コンテスト

- 競技内容
  - 指定のFPGAボード上に
    指定のアプリケーションを処理可能な
    計算機システムを実装し処理時間を競う
  - 指定FPGAボード
    - Digilent Atlys (Xilinx Spartan6 LX45)
    - Terasic DE2-115 (Altera Cyclone 4) など

- 結果
  - 第１回：準優勝
    - 自作MIPSコア+アクセラレータ２種
      - 高前田（山崎） 伸也, 吉瀬 謙二: メモリ抽象化フレームワークPyCoRAMを用いたソフトプロセッサ混載FPGAアクセラレータの開発
  - 第２回：３位
    - Microblaze+アクセラレータ2種
      - 田ノ元 正和, 枝元 正寛, 竹内 昌平, 高前田（山崎） 伸也: IPコア開発フレームワークPyCoRAMを用いたHW/SW協調FPGAアクセラレータの開発

ベンチマークアプリケーション4種

アプリケーション	説明	メモリシステムへの要求
310_sort	整数ソート	低レイテンシ
320_mm	行列積（整数）	高バンド幅
330_stencil	ステンシル計算（9点・整数）	高バンド幅
340_spath	最短経路問題	低レイテンシ

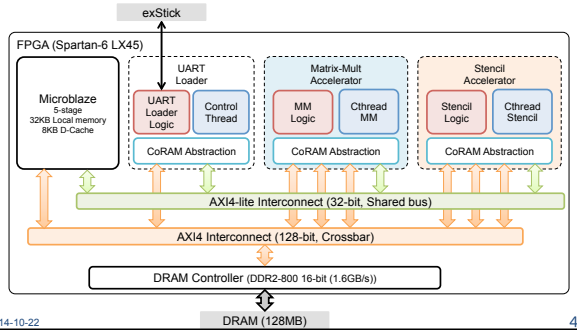
行列積・ステンシル計算 → 専用アクセラレータ向き

ソート・最短経路問題 → 専用アクセラレータは面倒

CPUコア+専用アクセラレータが良さそう

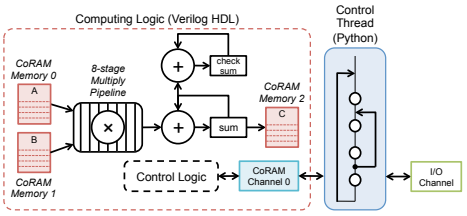
第2回コンテスト用アクセラレータ構成

- PyCoRAMでIPコア2種を実装
  - MicroblazeからはAXI4スレーブ経由でメモリマップドI/Oで制御



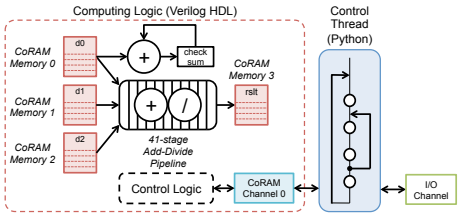
行列積アクセラレータ

- 行列A・B・Cの各行をCoRAMメモリに格納
  - DRAMとの間のデータ転送をコントロールスレッドが担当
  - 毎サイクル乗算パイプラインにデータを投入
  - 行列Bの転送と演算をダブルバッファリング
  - SIMD幅をメモリバンド幅を使い切るようにチューニング



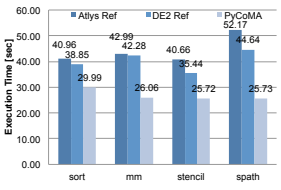
ステンシル計算アクセラレータ

- 元配列の3行と結果配列の1行をCoRAMメモリに格納
  - DRAMとの間のデータ転送をコントロールスレッドが担当
  - 毎サイクル加算・除算パイプラインに3点分のデータを投入
    - 過去3サイクルの入力値の合計を9で割る（3点×3サイクル=9点）
  - 計算1行分毎に結果を書き戻して元配列の次の1行を読み込む

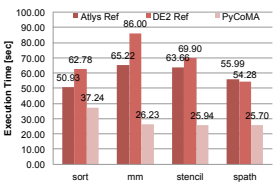


性能（第２回コンテスト）

- データ転送時間を含む総時間
- リファレンスデザインと比較して大幅な速度向上
  - 行列積・ステンシル・最短経路（ソフト実装）では データ転送に要する時間が殆ど
    - 僅差だった･･･？☺



リファレンスデータ



第１回コンテスト決勝データ

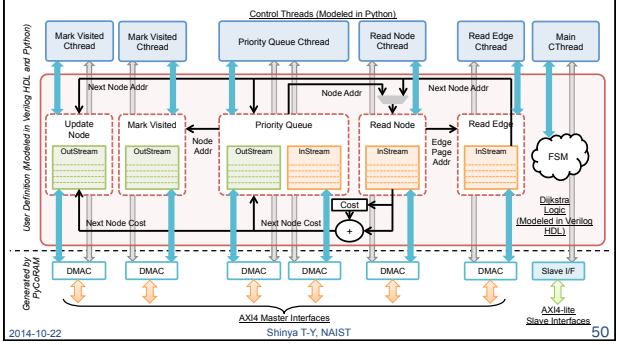
応用その１ グラフ処理 アクセラレータ

ダイクストラ法アクセラレータ[高前田+,CPSY2014-07]

- 不規則なメモリアクセスパターンを持つアプリにおける PyCoRAM適用可能性を明らかにする
  - 規則的なメモリアクセスパターンを持つアプリケーション（行列積・ステンシル）はバンド幅律速
  - メモリアクセスレイテンシの影響が大きいアプリで使えるの？
    - まずは実装してみましょう
- 今回の題材：グラフ処理
  - 最短経路探索（ダイクストラ法）
    - ボトルネックになりそうな箇所
      - 未訪問ノードの管理→距離をキーとした優先度キュー
      - 隣接ノード情報（コスト・親ノード）の読み書き→ページング

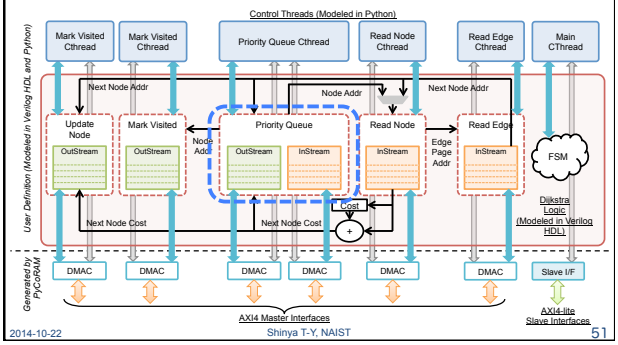
PyCoRAMを用いたダイクストラIPコア

- PyCoRAMを使って演算モジュールはVerilog HDLで実装
  メモリアクセス制御はPythonで実装



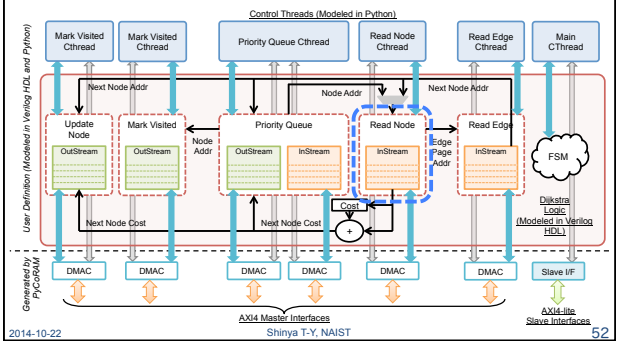
PyCoRAMを用いたダイクストラIPコア

- ステージ1: 最小コストノード取り出し



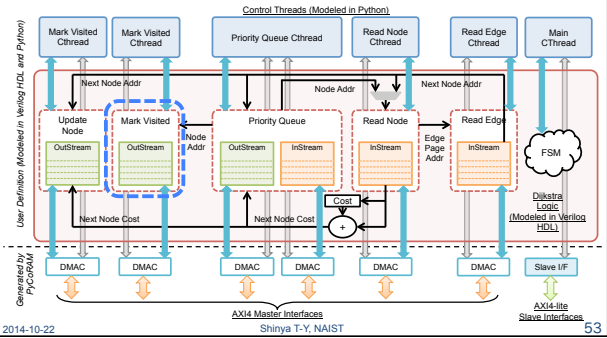
PyCoRAMを用いたダイクストラIPコア

- ステージ2：ノード情報読み出し



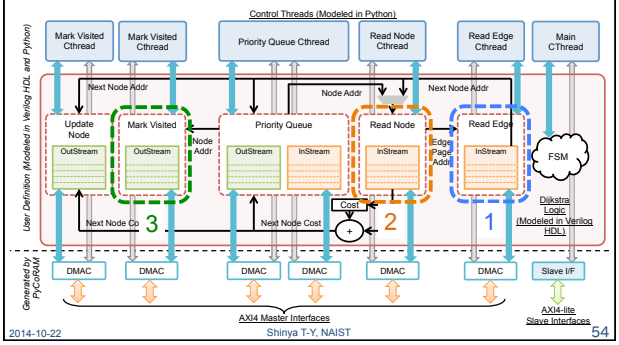
PyCoRAMを用いたダイクストラIPコア

- ステージ3：ノードに訪問済みフラグ書き込み



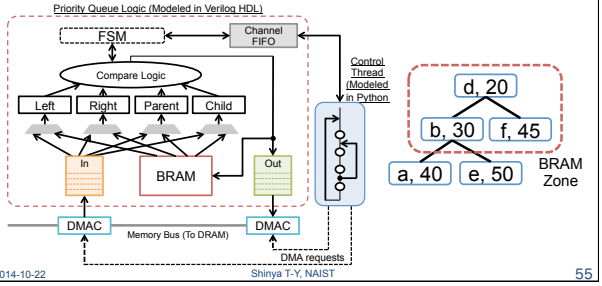
PyCoRAMを用いたダイクストラIPコア

- パイプライン動作：（1）エッジ読み出し→（2）隣接ノード読み出し→（3）隣接ノード更新



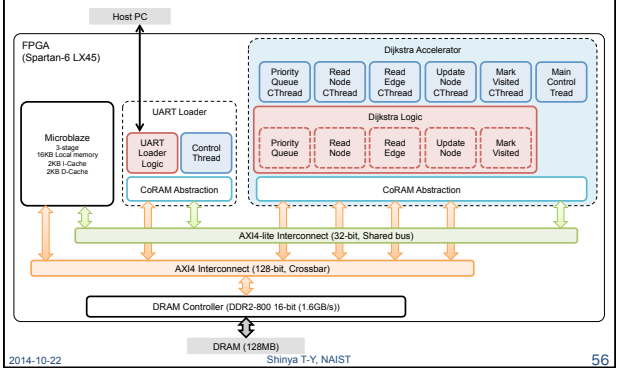
優先度付きキュー（ヒープ）

- CoRAMメモリ x2 + BRAM x1
  - 外部から読み出すための CoramInChannel
  - 外部へ書き込むための CoramOutChannel
  - コストが小さいノード群を格納する BRAM



評価環境：FPGAシステム

- ホストからUART経由で制御・グラフ構築にMicroblaze



実装の詳細

- AXI4インターコネクト：4構成
  - クロスバー2種：パイプラインレジスタ等を持つ高性能タイプ
    - C128: 128ビット幅
    - C32: 32ビット幅
  - 共有バス2種：リソース使用量を優先した省エリアタイプ
    - S128: 128ビット幅
    - S32: 32ビット幅
- AXI4バスでは異なるマスターポート間では Read/WriteのIn-order順番が保証されていない
  - 先にバスにリクエストが発行したからといって 必ず先に処理されるわけではない
  - 特に Write → Read の依存関係には注意が必要
  - 解決策
    - AXI4バスの設定で書き込みポートのPriorityを高くする

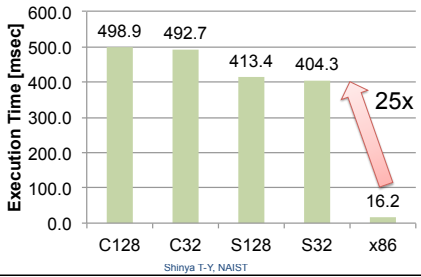
評価

- FPGAボード実機で評価
  - ボード: Digilent Atlys
    - FPGA: Spartan-6 LX45
    - DRAM: DDR2-800 (1.6GB/s), 128MB
  - ツール: Xilinx PlanAhead 14.7, XPS 14.7

- 汎用PC上と比較
  - Intel Core i7 3770K (3.5GHz), DDR3-1600 (12.8GB/s ×2)
  - Linux (Ubuntu 14.04), gcc 4.8.2 (-O3)
- グラフ
  - XORSHIFT乱数を用いてランダムに生成
  - ノード数: 5000, エッジ数: 100000
    - より大規模なグラフはデバッグが間に合わなかったので 今後の課題ということで・・・

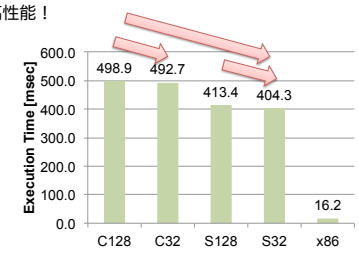
実行時間

- FPGA上の実装は汎用PCと比べて25倍程度低速☹
  - メモリバンド幅あたりの性能でも1.5倍程度悪い・・・
  - なぜか？
    - データセットが小さい・OoOプロセッサのMLP抽出能力は凄い



実行時間

- FPGAでの実行時間を比べてみると直感と真逆の結果
  - クロスバーよりも共有バスの方が高性能！
  - バス幅が狭い方が高性能！
- なぜか？
  - 共有バスの方が レイテンシが短い
  - バス幅が短い方が レイテンシが短い
  - ⇨ 必要なモノ： 高バンド幅ではなく 短レイテンシ



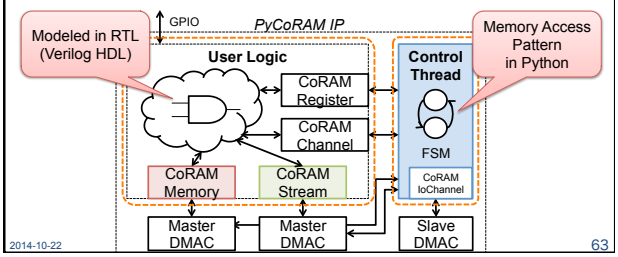
まとめ

今後の展望

- ロジックのモデリング方式
  - コンピューティングロジックもHDLで書きたくない
  - 計算カーネルに特化したマルチパラダイム処理系・DSL？
    - MyHDLとの連携方式を実装中
- 静的解析による性能・電力チューニング
  - より少ないエフォートでより高いメモリ性能
  - 電力効率の高いネットワーク・システム構成の自動選択
- 実アプリケーションへの適用
  - イーサネット直結系のオフローダ（ストリーム処理？）
  - ストレージ直結系のアクセラレータ（ビッグデータ？）

まとめ

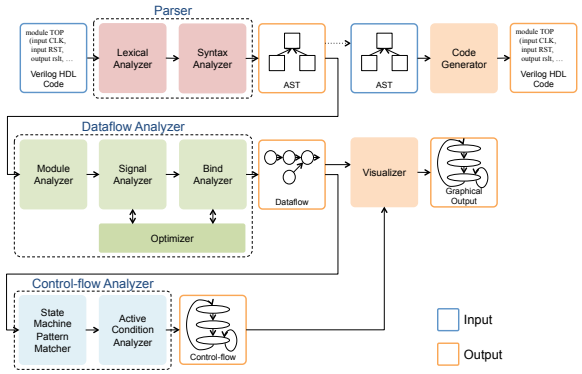
- PyCoRAM: 高位合成技術とメモリシステム抽象化を用いたアクセラレータIPコア開発のためのフレームワーク
- 開発したツール・フレームワークはgithubにて公開中
  - PyCoRAM: http://shtaxxx.github.io/PyCoRAM/
  - Pyverilog: http://shtaxxx.github.io/Pyverilog/



PyCoRAMの実装と動作環境

- ピュアなPythonによる実装
  - 全体で4947行で実装できたのでコンパクト！
    - Python-Verilogコンパイラ: 3828行
    - RTLトランスレータ: 769行
  - Verilog HDLのRTL静的解析にはPyverilogを利用
    - Python-based Hardware Design Processing Tool-kit for Verilog HDL
    - 全体で12665行とかなりコンパクト
  - コード生成にはJinja2 (テキストテンプレートエンジン)を利用
    - テンプレート用テキストファイルが8000行くらい
- 動作環境：LinuxかMacを推奨
  - Python 3.3 (or later)
  - Icarus Verilog 0.9.6 (or later)
  - Jinja2 (2.7 or later)

Pyverilog: Python-based Hardware Design Processing Tool-kit for Verilog HDL



Visualizerの出力