Approaches for Heterogeneous Systems

Nuno Paulino
DEI, FEUP, 2022

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

INESCPORTO
INSTITUTO DE ENGENHARIA DE SISTEMAS E COMPUTADORES DO PORTO

Summary

1. *Generation of Custom Run-time Reconfigurable Hardware for Transparent Binary Acceleration (PhD)*
   a. *Custom Loop Accelerator*
2. K-means on FPGA via OpenCL (IEEE Access)
3. Recent stuff: Binary Translation Framework and CrispyHDL

Generation of Custom Run-time Reconfigurable Hardware for Transparent Binary Acceleration

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO
INESCPORTO
INSTITUTO DE ENGENHARIA DE SISTEMAS E COMPUTADORES DO PORTO

Context

- *Generation of Custom Run-time Reconfigurable Hardware for Transparent Binary Acceleration*
  - Topic of my PhD thesis, 2011 - 2015
  - In summary: a work about translating sequences of instructions from MicroBlaze into accelerator circuits

- Why?
  - General methodology to reduce power consumption and improve performance in embedded applications

50 Years of CMOS Processor Technology

- Dennard Scaling
  - Scale down
  - Voltage down
  - MHz up
  - Heat dissipation → constant

- Too small → current leakage!
- 2005 → End of Single-core scaling

- How far can Multi-Core go?
  - Dark Silicon
  - Amdahl's Law

15 Years of incremental improvements…

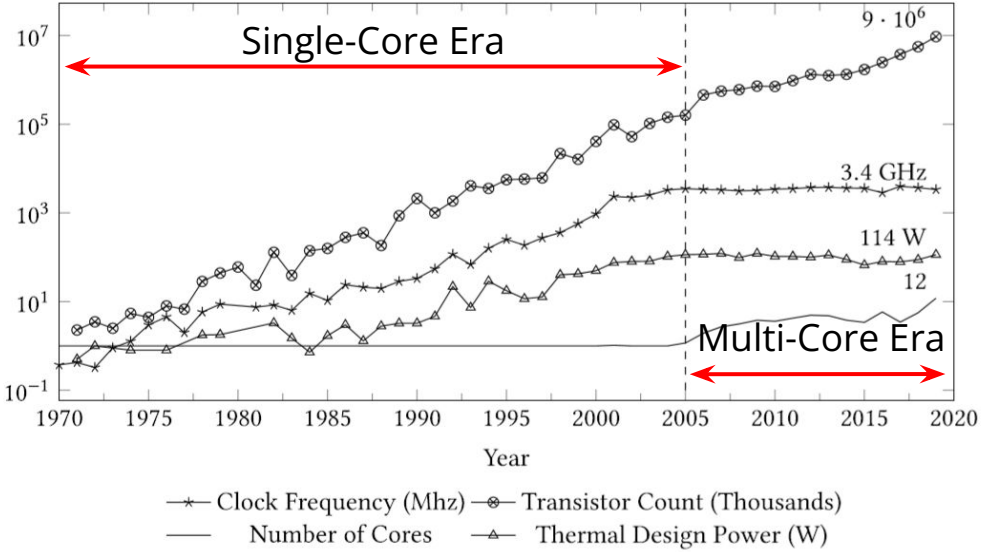


Fig. 1. Trends for desktop and server grade processors throughout the last 50 years, built from 950 data points from CPU DB [23] and Intel's and AMD's product pages

Improving Performance?

- Approaches to improve performance?

1. Improve sequential processors
    a. Superscalar (dynamic)
    b. VLIW (static)
    c. Multi-core

Architecture and technology limitations

2. **Heterogeneous** architectures
    a. Processor + GPUs
    b. System-on-a-chip
    c. Workload specific circuits

Problems:
- Laborious hardware design
- Difficult to adopt and maintain
- Expensive to produce

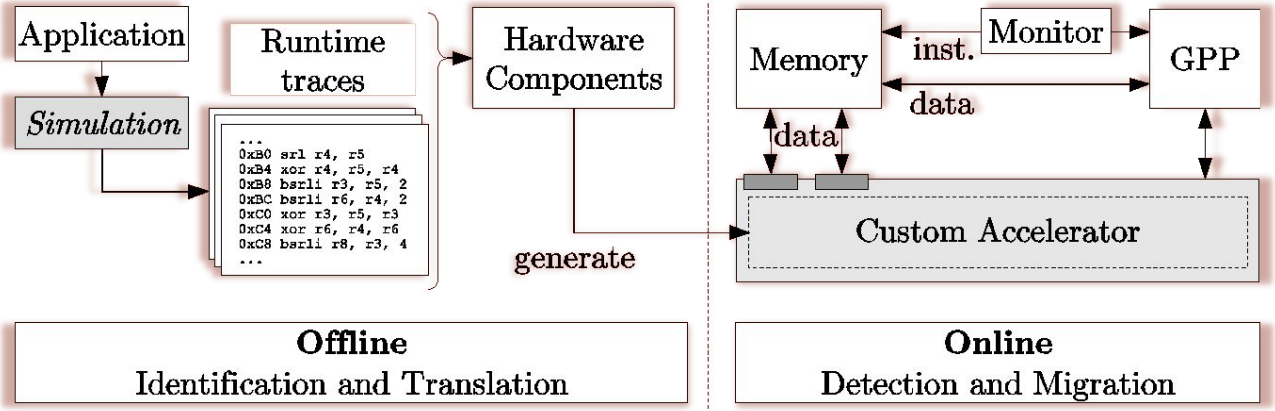
How to automate generation of specialized reconfigurable accelerators for embedded applications?

Objectives

1. Design an accelerator architecture capable of:
    a. Executing loops, exploiting **ILP** and **loop pipelining**
    b. Exploiting **data parallelism** with parallel accesses to data memory
2. Generate instances from instruction traces
3. Automatically transfer control from CPU to Accelerator
4. Augment the accelerator with Dynamic Partial Reconfiguration

The work would target FPGAs as the device, and Xilinx's **MicroBlaze** processor as the host CPU

General Approach

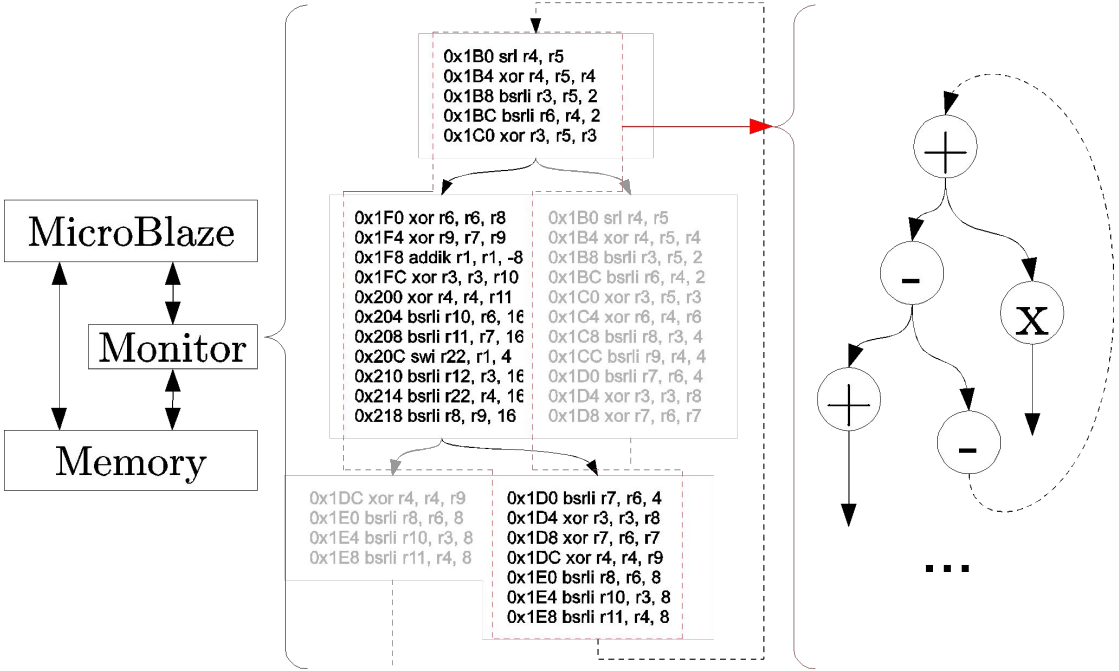


1. Identify frequent binary loop traces (existing work)
2. Translate loops into hardware accelerators
3. Detect imminent execution of loops at runtime
4. Migrate execution to accelerators

Extracting Trace Loops

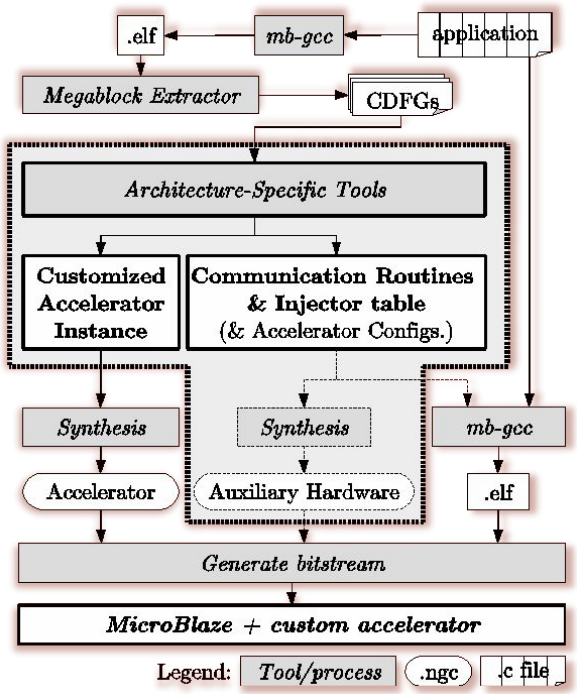
Megablocks

- Instruction traces
- Repeating
- One entry, many exits
- Represented as CDFGs to expose parallelism



Toolflow

1. Simulate execution
   a. extract traces
   b. choose traces
2. Generate accelerator instance
   a. Schedule operations
   b. Generate verilog
   c. Communication code
3. Synthesis of CPU + Accelerator
4. **Execution!**



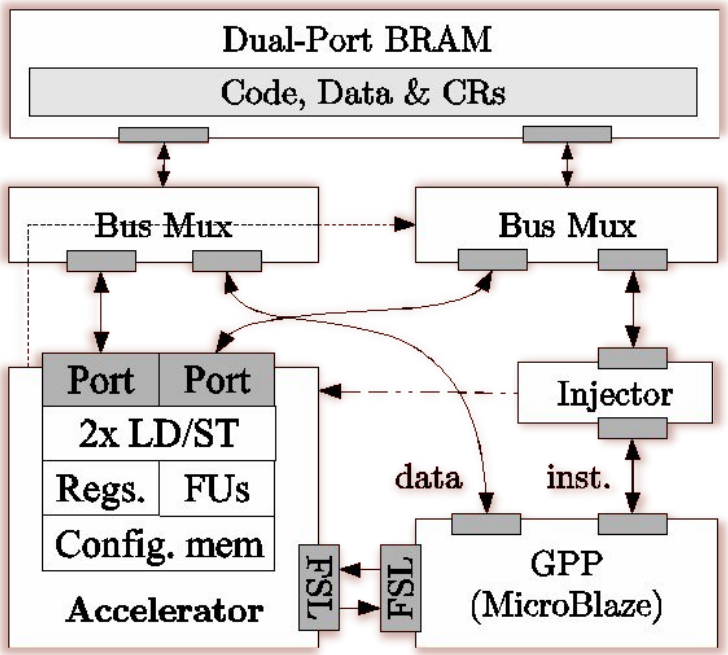
Toolflow (Extras)

- **What tools were used/developed?**

- Megablock Extractor
  - A Java tool by Prof. João Bispo (based on a simulator from INESC ID) as part of his PhD
- Design of the loop accelerator
  - Multiple designs I made in Verilog, synthesized with Xilinx EDK (defunct tool)
  - Simulated/debugged in ISim
- CDFG Scheduler
  - Tool written from scratch in C
  - Re-implemented in MATLAB (!) to test modulo scheduling code
  - Re-re-implemented in C again…
  - Abandoned (?)… on course to be re-re-re-implemented in new code base
- Communication Routine generator
  - Another separate tool in C
  - Integrated into the scheduler eventually
  - Capable of generating different types of routines based on system architecture details…

System Architecture

- MicroBlaze processor
- Loop accelerator instance
- Injector module
- Shared data memory
  - Bus muxes to share the memory



N. M. C. Paulino, J. C. Ferreira and J. M. P. Cardoso, "Generation of Customized Accelerators for Loop Pipelining of Binary Instruction Traces," in IEEE Trans. VLSI 2017
https://ieeexplore.ieee.org/document/7506263

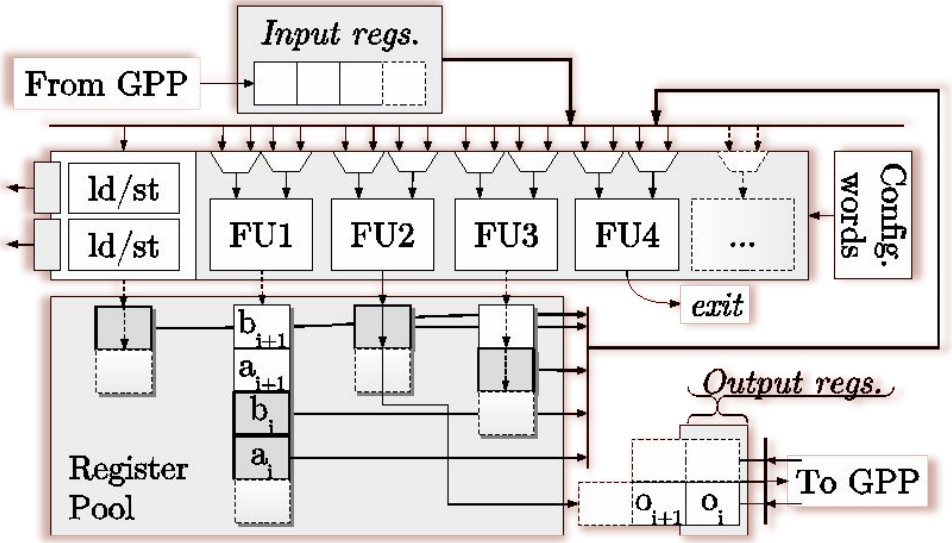
Loop Accelerator Architecture

Structure

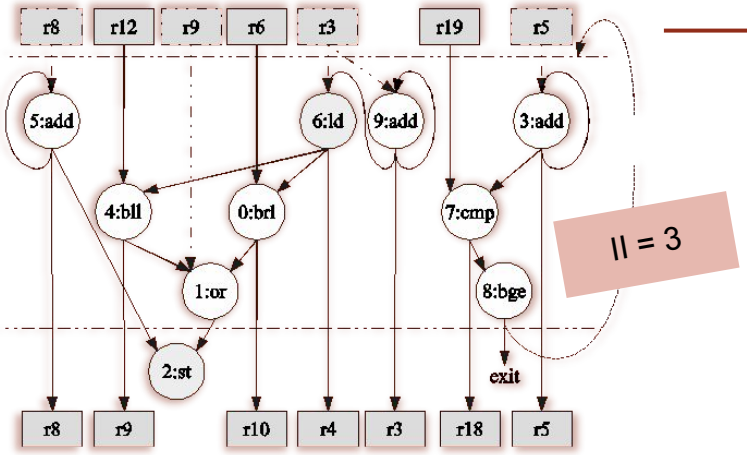
- One row of units in parallel
- Specialized interconnections
- Configuration memory per cycle control

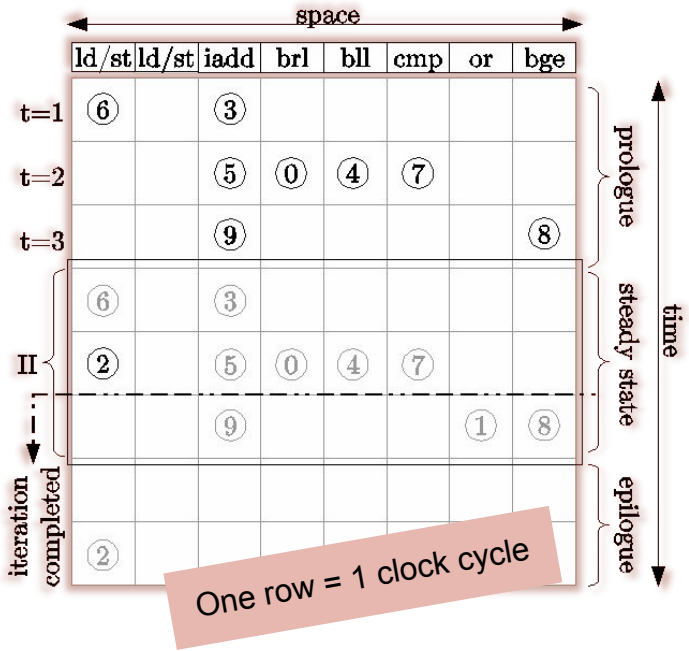
Features

- Loop-pipelining
- Floating-point operations
- Loop-specific units and connections



Modulo Scheduling





- Add units to guarantee minimum II
- Scheduling resource constrained to two ports
- Create multiplexers after scheduling

U.PORTO FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

INESCPORTO INSTITUTO DE ENGENHARIA DE SISTEMAS E COMPUTADORES DO PORTO

Experimental Results - Speedups

Setup

- VC707 Board (Virtex-7 xc7vx485)
- 13 float and 11 integer kernels
  - Avg. 33 instruction in each loop
- Baseline: MicroBlaze @ 110 MHz

Accelerator + Microblaze vs Baseline

- Geomean: 6.60x for integer set, 4.61x for floating-point set

Resource Requirements

- 1.13x the FFs, and 1.83x the LUTs a MicroBlaze requires

~4x faster, ~2x "larger"

Baseline vs. ALU-based loop accelerator

- 2 ALU accelerator: 2.1x
- 4 ALU accelerator: 3.5x
- 8 ALU accelerator: 4.1x

Custom accelerator 1.78x faster than 4-issue VLIW, for 20% the LUTs

Baseline vs. VLIW Cores

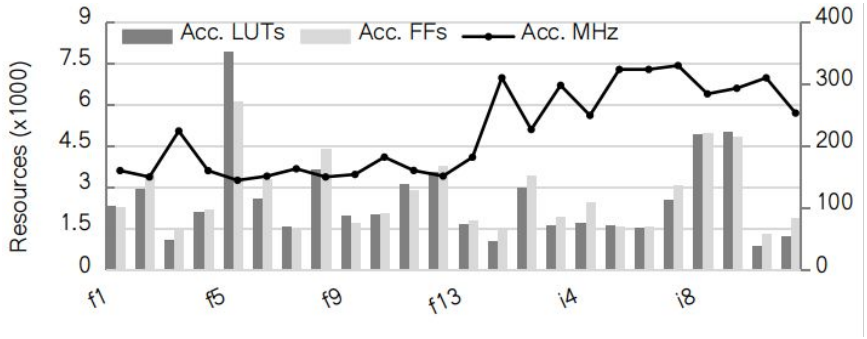
- 2-issue: 2.2x
- 4-issue: 2.5x
- 8-issue: 2.6x

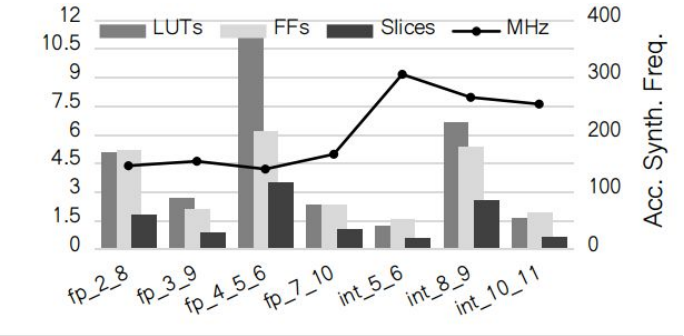
Custom accelerator ~= as fast as ALU based instance, for 0.5x the slices

U.PORTO FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

INESCPORTO INSTITUTO DE ENGENHARIA DE SISTEMAS E COMPUTADORES DO PORTO

Experimental Results - Resources





Single loop instances (1 config)

- Number of FUs minor impact
- Bigger config. memory → more resources
  - Specially LUTs (distributed RAM used to implement very wide word memories)

Multiple loop instances (+1 config)

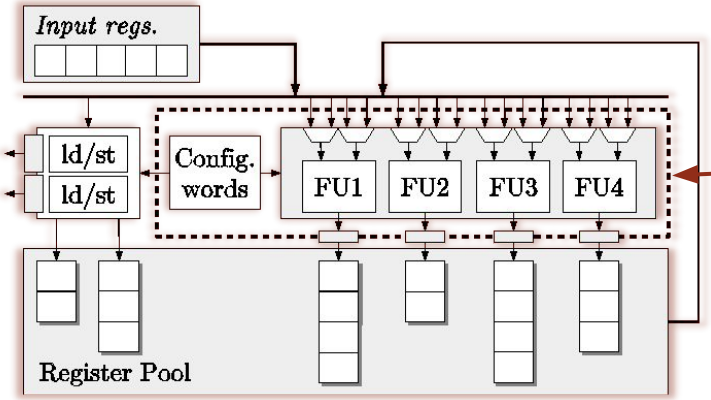
- Even bigger config. memory
- More FUs and muxes
  - Frequency drop
  - Very long synthesis times

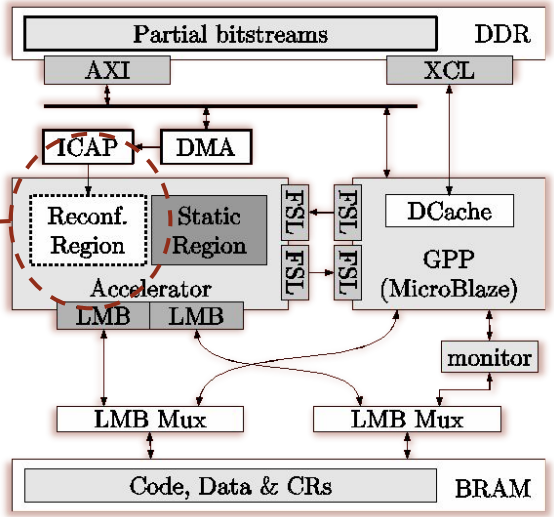
U.PORTO FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

INESCPORTO INSTITUTO DE ENGENHARIA DE SISTEMAS E COMPUTADORES DO PORTO

Adding DPR

- DPR → Change a region of the FPGA configuration at runtime
  - **Reutilize resources** by changing the Functional Units, config memory, and muxes





Experimental Results - DPR

Setup

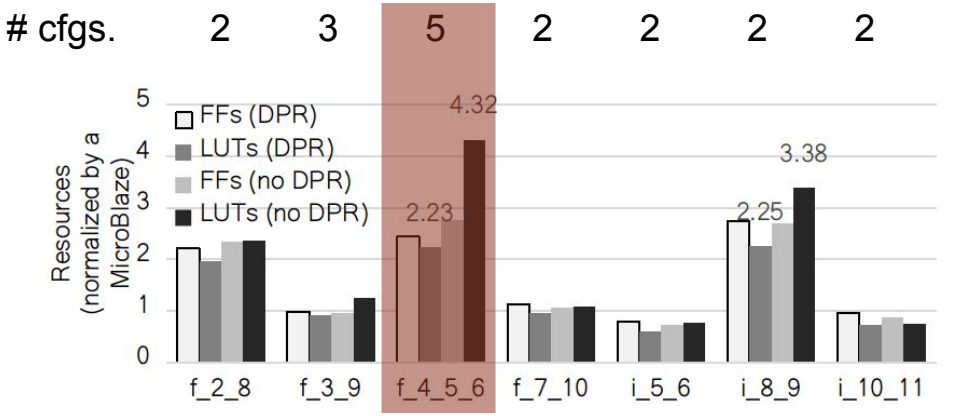
- 13 float and 11 integer kernels
  - 7 accelerators (2 to 5 cfgs.)

- Local memory w/code and data
- External memory w/partial bitstreams
- DMA-driven ICAP reconfiguration

Speedups and Overhead

- 4.2x (fp), 2.6x (int)
- DPR time: 3ms

Speedup decrease by 30%...



- DPR best for more configs
- f456 → 0.5x LUTs and 0.8x vs no-DPR

- Time for 10 configs (synthesis)
  - 9.6 min (DPR) vs. 10.8 hrs (no-DPR)

U.PORTO FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

INESCPORTO INSTITUTO DE ENGENHARIA DE SISTEMAS E COMPUTADORES DO PORTO

Conclusion

- A flow for generating instances of a type of accelerator design
- Using instruction traces from simulation
- Validated on-chip, achieving speedups vs CPU-only

- Problems
  - Usability for the future
  - Using with different CPUs?
  - Exploring different accelerator designs (e.g., CGRAs)
  - Executing “real” applications, not just kernels
  - Doing it all at runtime on-chip

3.K-means on FPGA via OpenCL

3. K-means on FPGA via OpenCL

- This work follows a “traditional” approach to heterogeneous systems:
  - CPU side (C/C++) code + using APIs like OpenGL, OpenCL, or OpenMP to communicate and dispatch workload onto a GPU (commonly)
  - For some time now, **OpenCL compilation for FPGAs** has being adopted/developed
    - Xilinx does this by lowering C/C++ to LLVM, and then to RTL
    - The RTL obeys certain interfaces that make it compatible with OpenCL APIs

- *Optimizing OpenCL Code for Performance on FPGA: k-Means Case Study With Integer Data Sets* https://ieeexplore.ieee.org/document/9170625

Objectives

- Study a use case of HLS for FPGAs using OpenCL
  - Specifically, evaluate performance and design effort of Xilinx OpenCL HLS (SDAccel)

Now part of Xilinx Vitis

- Outperform a sequential CPU execution of k-means
  - When executing k-means as C on CPU
  - When executing k-means as OpenCL kernel on CPU

- Compare runtime, power consumption, and power/performance tradeoff

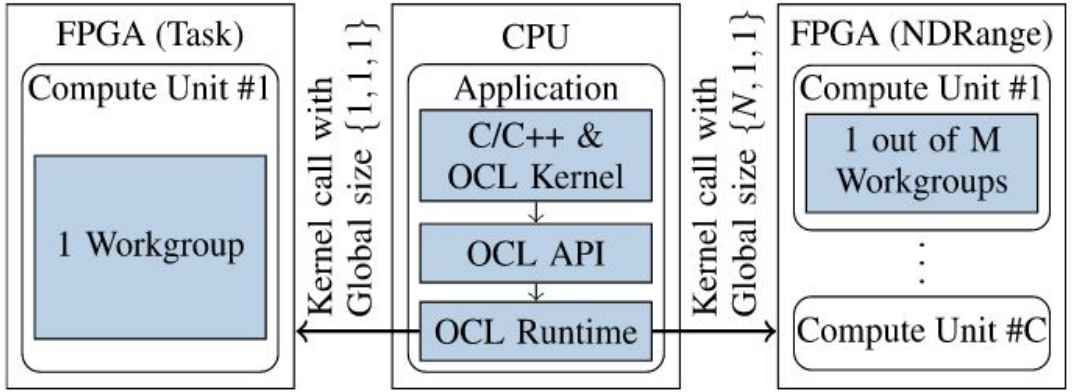
k-means Algorithm

- From a given set of initial cluster centroids:
  a. for each point, compute distance to all centroids
  b. assign each point to its closest centroid
  c. compute new centroids based on point assignments
  d. repeat from “a” until centroids converge (to a given tolerance)
- What is the best way to parallelize?

Algorithm 1 k-means Clustering

```
Data: Set of N = X_1, X_2, ..., X_N input data, where
      X_n = x_1, x_2, ..., x_d, threshold, K, D, N
Result: Set of K cluster centroids C = C_1, C_2, ..., C_k
        and assignments of each datum X_n to a cluster k
while error > threshold do
    set old_error = error;
    set error = 0;
    forall the X_n in X do
        set mindist = 0;
        forall the C_k in C do
            Compute distance dist of X_n to C_k;
            if dist < mindist then
                mindist = dist;
                assign X_n to cluster k;
    error = error + mindist;
    forall the C_k in C do
        Compute new C_k from points assigned to cluster
        k
```

OpenCL Workgroup Computing Model



OpenCL Task-Kernel vs NDRange Kernel execution; for NDRange, workgroups have local size {1 < n < N, 1, 1}, where N = total #workitems

Example OpenCL: dot product

- Outer loops typically disappear
- They become “workgroups”, and iterations become “workitems”
- Workgroups execute in parallel

data in shared global memory (i.e., DDR)

```
__kernel void DotProduct (__global float* a, __global float* b, __global float* c, int iNumElements)
{
    // find position in global arrays
    int iGID = get_global_id(0);

    // bound check (equivalent to the limit on a 'for' loop for standard/serial C code
    if (iGID >= iNumElements)
    {
        return;
    }

    // process
    int iInOffset = iGID << 2;
    c[iGID] = a[iInOffset] * b[iInOffset]
              + a[iInOffset + 1] * b[iInOffset + 1]
              + a[iInOffset + 2] * b[iInOffset + 2]
              + a[iInOffset + 3] * b[iInOffset + 3];
}
```

workitem ID

Baseline OpenCL

- Straight C → OpenCL conversion
- Purely sequential
  - In OpenCL, its classified as a **“task-kernel”**
  - Does not exploit workgroup model

- In this case
  - FPGA can explore deep hardware pipelining, where CPU cannot
  - **One compute unit is instantiated on the FPGA**

```
__kernel void s1kmeans1(global uint *data, int n, int m,
int k, int t, global uint *centr, global int *labels,
global uint *c1, global int *counts, global int *itcount)
{
  ulong old_error, error = INT_MAX;
  uint i = 0, j = 0; itcount[0] = 0;

  do {                                                    A
    old_error = error, error = 0; // save error
    for (i = 0; i < k; i++) {
      counts[i] = 0;  // clear tmp counts
      for (j = 0; j < m; j++) c1[i*m+j] = 0;
    }

    for (int h = 0; h < n; h++) {                         B

      uint mindist = INT_MAX;                             C
      for (i = 0; i < k; i++) {

        ulong dist = 0, diff = 0;                         D
        for (j = 0; j < m; j++) {
          diff = data[h*m+j] - centr[i*m+j];
          dist += diff*diff;
        }

        if((int)(dist/2) < (int)(mindist/2)) {
          labels[h] = i;
          mindist = dist;
        }
      }

      counts[labels[h]]++;
      for (j = 0; j < m; j++) // new aux sum
        c1[labels[h]*m+j] += data[h*m+j];

      error += mindist; // update error
    }

    itcount[0]++;
    for (i = 0; i < k; i++) // new centroids
      for (j = 0; (j < m) && (counts[i] > 0); j++)
        centr[i*m+j] = c1[i*m+j] / counts[i];

  } while(abs((error - old_error)) > t);

}
```

Optimizations

Kernel	Description
v1	Task-kernel; Baseline code
v2/v3	Task-kernel; v1 + specialization for $D = 8$ or $D = 16$
v4	*NDRange*; Computation of new centroids by host
v5	*NDRange*; v4 + specialization for $D = 8$
v6	*NDRange*; Only one point computed per work-group
v1b	v1 + burst access optimization
v5b2	v5 + burst access optimization, specialized for $D = 2$
v5b8	v5 + burst access optimization, specialized for $D = 8$
v5b16	v5 + burst access optimization, specialized for $D = 16$

Different tested k-means kernel versions

C

```
uint mindist = INT_MAX;
for(i = 0; i < k; i++) {
```

D

```
  uint8 d = data[h] - centr[i];
  uint dist =
   d.s0 * d.s0 + d.s1 * d.s1 +
   d.s2 * d.s2 + d.s3 * d.s3 + d.s4 * d.s4 +
   d.s5 * d.s5 + d.s6 * d.s6 + d.s7 * d.s7;

  (...) // compare dist with mindist
}
```

Excerpt from v2

Removal of one inner w/ 8 iterations loop using a vector datatype of 8 elements

- In this case
  - Vectorization removes on inner loop
  - We confirmed that Intel's OpenCL runtime performs auto-vectorization

Optimizations - v4/v5

- Workgroup model
  - “Normal” for OpenCL workloads
  - Nr workgroups determined by max. workgroup size and total nr. of workgroups
  - Workgroups → parallel
- In this case
  - CPU explores parallel work groups due to independent data
  - But FPGA can **in addition** explore pipelining of inner loops
  - **Multiple compute units are instantiated on the FPGA**

```
__kernel void s1kmeans4(
global uint *data, int n, int m, int k, float t,
global uint *centr, global int *labels,
global uint *mindist)
{
  size_t gsz0 = get_global_size(0U);
  size_t gid0 = get_group_id(0U);
  int offset = gid0 * (n/gsz0);

  int h;
  for (h = offset; h < offset + (n/gsz0); h++) {

    mindist[h] = INT_MAX;
    for (int i = 0; i < k; i++) {

      uint dist = 0;
      for (int j = 0; j < m; j++) {
        uint diff = data[h * m + j]
                    - centr[i * m + j];
        dist += diff*diff;
      }

      if ((int)(dist/2) < (int)(mindist[h]/2)) {
        labels[h] = i;
        mindist[h] = dist;
      }
    }

  }

}
```

B

C

D

- Loop A moved to host side (not very paralellizable)
- Loop B bounds modified based on workgroup size

Optimizations - v5b

- Workgroup model with burst memory access inference
  - **Loop E3 - Burst read points**
  - Uses more device BRAM
  - Explicit local multi-port memories load up to TMPPTS points
    - *TMPPTS* could have been larger, up to device limits

```
uint tmplabels[MAXPTS], tmpdist[MAXPTS]                    E1
__attribute__(xcl_array_partition(cyclic,16,1));
uint8 tmppts[TMPPTS], tmpcentr[8 * MAXK]
__attribute__(xcl_array_partition(cyclic,2,1));
```

(E2 - loop for burst reading into “tmpcentr” omitted)

```
int ptctr = TMPPTS;                                         B
for(int h = 0; h < npoints; h++) {
  if(ptctr == TMPPTS) {                                     E3
    ptctr = 0;
    for(int j = 0; j < TMPPTS/2; j++) {
      int idx = ((offset + h)/2) + j;
      uint16 tmpread = data[idx];
      tmppts[(j*2)+0] = tmpread.lo;
      tmppts[(j*2)+1] = tmpread.hi;
    }
  }

  (...) // for every centroid                               C
    // adapt D segment in kmeansv2/v3                       D
    // to resort to "tmppts" and
    // "tmpcntr" to compute distances
    (...) // compare dist with mindist
  ptctr++;
}
```

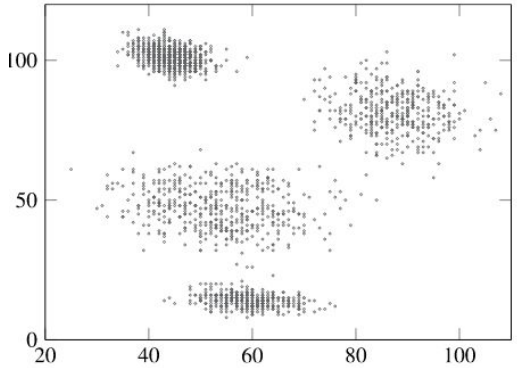
(E3 - loop for burst writing into outputs omitted)

Excerpt from v5b

Experimental Setup

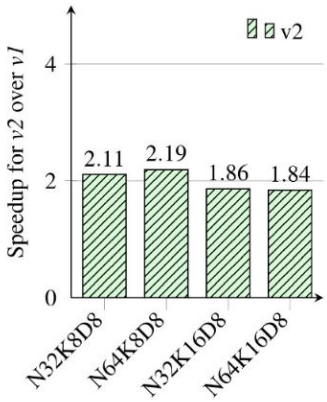
- Desktop CPU
  - Intel Core i7-6700K CPU (4 GHz)
  - Alpha Data ADM-PCIE-KU3
    - Kintex-6 XCKU060 FPGA
  - 32 GB RAM
- Execution
  - Host allocates input/output memory
  - Initial centroids computed using *kmeans++*
  - OpenCL API using Xilinx's runtime for FPGA target, or Intel's runtime for CPU
- Data
  - Generated synthetically by our own randomly correlated cluster generator

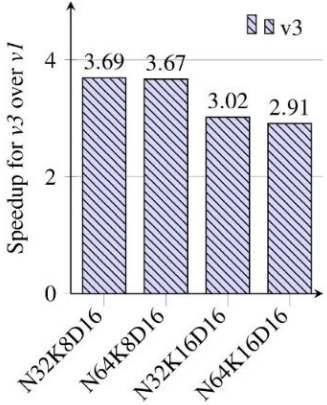
Alpha Data ADM-PCIE-KU3

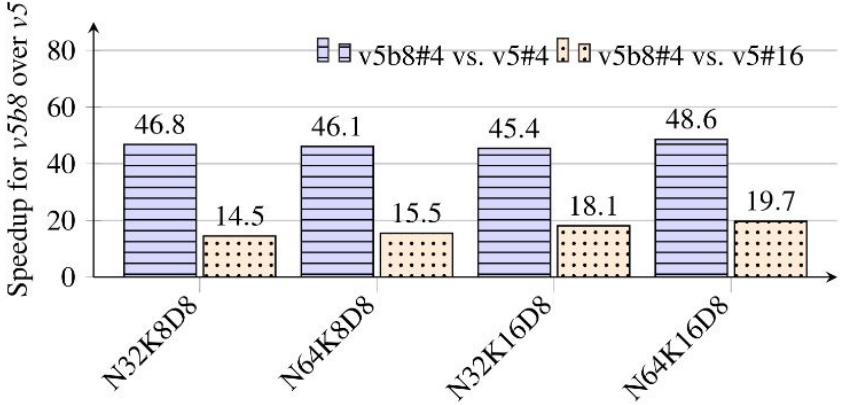


Example dataset generated for D = 2, K = 4, N = 4k

Experimental Results – Performance on FPGA







Speedup of vectorization alone vs OpenCL baseline (v1), on FPGA

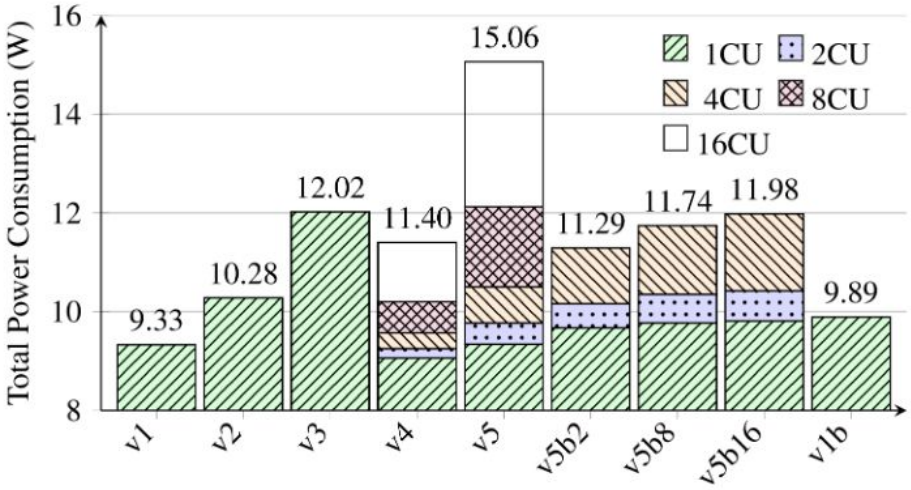
- i.e., task kernels w/ and w/o vectorization

Speedup of burst access over analogous versions (e.g., v5b over v5)

- Workgroup kernels w/ vectorization, w/ and w/o burst accesses

Experimental Results - Power on FPGA

- Power measured from post-route reports
  - For all code variants
  - For different numbers of compute units (where applicable)
- The best performing versions (v5b) only support up to 4 compute units
  - (Lack of FPGA resources)



Power consumption on FPGA for all cases and different numbers of CUs

Experimental Results - Summary FPGA vs CPU

- Power measured on CPU using RAPL interface
- Compared best performant code version per device, per problem size

{N,K,D}	Best Version CPU	Best Version FPGA	Speedup	Power (W) CPU	Power (W) FPGA	Energy (J) CPU	Energy (J) FPGA
{32,8,2}	v1#1	v5b2#4	0.83	≈40	11.29	1.32	0.45
{64,8,2}	v6#8		0.86			0.69	0.23
{32,16,2}	v6#2		0.75			1.69	0.63
{64,8,2}	v6#8		0.75			2.28	0.86
{32,8,8}	v2#1	v5b8#4	0.76	≈40	11.74	0.65	0.25
{64,8,8}	v2#1		0.78			0.86	0.33
{32,16,8}	v6#4		**1.54**			1.06	0.20
{64,16,8}	v2#1		**1.16**			4.32	1.09
{32,8,16}	v3#1	v5b16#4	0.81	≈40	11.98	0.33	0.12
{64,8,16}	v3#1		0.76			0.55	0.22
{32,16,16}	v3#1		**1.50**			1.38	0.28
{64,16,16}	v6#4		**1.44**			1.46	0.30

FPGA Wins!

FPGA Wins!

U.PORTO FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

INESCPORTO INSTITUTO DE ENGENHARIA DE SISTEMAS E COMPUTADORES DO PORTO

Experimental Results - Observations

Some points that affect performance

- Type of kernel (task vs NDRange)
- Number of Compute Units
- Effect of data set parameters (N, K, D)
- Loop pipelining and vectorization
- Local memories (multi-port) and burst accesses
- Cost of CPU vs FPGA (performance vs power spent)

Experimental Results - Type of Kernel

- NDRange + Loop pipelining
  - Allows us to **explore the workgroup paradigm**
  - Combined with a workgroup size of {1, 1, 1}, each workgroup contains a **single fully pipelined loop** (benefit of burst accesses to memory)
  - We dispatch each workgroup into one Compute Unit (i.e., copy of the circuit on the FPGA)
  - Only possible in this case since no data dependencies between work-items!
    - Might not be the case for other kernels, but possible here, due to the code design
- Task kernel
  - Less efficient since the original code had 3 nested loops
  - We can't pipeline them all!

Experimental Results - #Units e Dataset

- More Compute Units
  - More workgroups being computed in parallel
  - The tools generate designs where the frequency doesn't change with the nr of units
    - (Probably due to efficiently isolating different blocks and preventing long wires)
- N, K, and D
  - D has generally no effect, just increases the number of work-items, or loop iterations
  - K and D affected the use of vector data types
    - Some implementations are specialized for a particular dimension, e.g., for D = 8
    - In this case, **that** particular version of the code **doesn't** support datasets where D is different from 8
    - That is: specialization *may* compromise general applicability

Experimental Results - Pipelining & Vectorization

- Vitis/Vivado HLS automatically pipelines loops, **if possible**
  - Iterations can contain function calls, which themselves must be pipelineable
    - e.g., cannot have arbitrary runtime
  - If outer loop, then must have fully unrollable inner loops
  - No accesses to high-latency memory
  - No data dependencies between iterations if better, for lower initiation interval
- Vectorization
  - Intel's OpenCL runtime applies vectorization automatically, i.e., no need to change code
  - Xilinx's HLS compiler **doesn't** -> need to change code, but more design freedom (?)
  - Useful for removing/simplifying loops where the trip count is a multiple of the vector width (if it isn't we can always pad the data with zeros)

Experimental Results - Local Memories & Burst

- Local multi-port memories
  - 1 Cycle access to multiple data items
  - Works better if accesses are **coalesced** (e.g., adjacent addresses)
  - Several strategies for memory partitioning
    - The “best” depends on the kernel (how we want to process the data)
  - Good results, but BRAMs are a “rare” resource inside (most) FPGAs
    - Bottlenecks to speedup are usually memory access related…
- Burst accesses
  - Follow the “recipe” so that the HLS compiler generates code for burst accesses
  - Multiple data items fetched per beat, multiple beats, one beat per cycle -> lots of data
  - We add more code (to fill the local memories) but the workload code iterates faster

Experimental Results - Cost!

- CPU
  - Intel Core i7-6700K (relatively high-end), 14nm
  - Release date 2014
  - **$450**
  - ~40W (measured for this code!)
- FPGA
  - Alpha Data ADM-PCIE-KU3 card (Xilinx Kintex UltraScale XCKU060-2, low/mid range)
  - Release Q2'2015
  - **$2700 (6x higher than CPU!)**
  - ~11W (measured for this code!)

But do the math!

- FPGA 1.5x faster in the best case; 4.8x fewer energy
- Cost is 6x, in exchange for 4.8x less energy, and 1.5x faster
  - 6 / 4.8 = 1.25x more costly (in the long term with saved energy)

Conclusions

- Mid-grade FPGA can outperform high-end CPU
  - Best version **725x faster** than OpenCL baseline on FPGA
  - But not without significant code changes, producing non-portable OpenCL code
  - CPU still faster in most cases, but best FPGA case outperforms CPU **by 1.5x** with **4.8x** lesser power
- Four public artifacts
  - An Implementation of K-means written in C -
    - https://codeocean.com/capsule/3208075/tree/v1
  - A Test Harness for Multiple OpenCL Implementations of the k-means Algorithm
    - https://codeocean.com/capsule/2348736/tree/v1
  - A Generator of Randomly Correlated N-Dimentional Clusters
    - 10.13140/RG.2.2.34866.43200
  - A Batch of Integer Datasets for Clustering Algorithms
    - 10.21227/smta-vv06

4.Binary Translation Framework

Binary Translation Framework

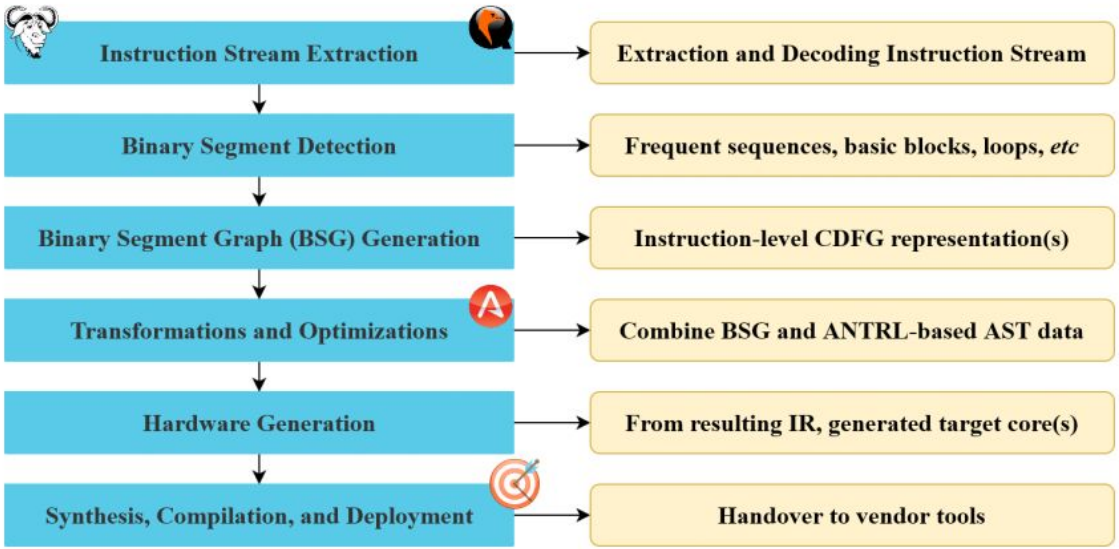
- Our own previous work:
  - Targeted only MicroBlaze G
  - Generated/supported only one specific type of pipelined loop accelerators
  - Functional (+), but limited (-)

How to explore hardware generation from trace/post-compile information for **more ISAs**, and targeting **more/different accelerator/core designs?**

The purpose of the Binary Translation Stack is to implement this flow.

Framework Stack

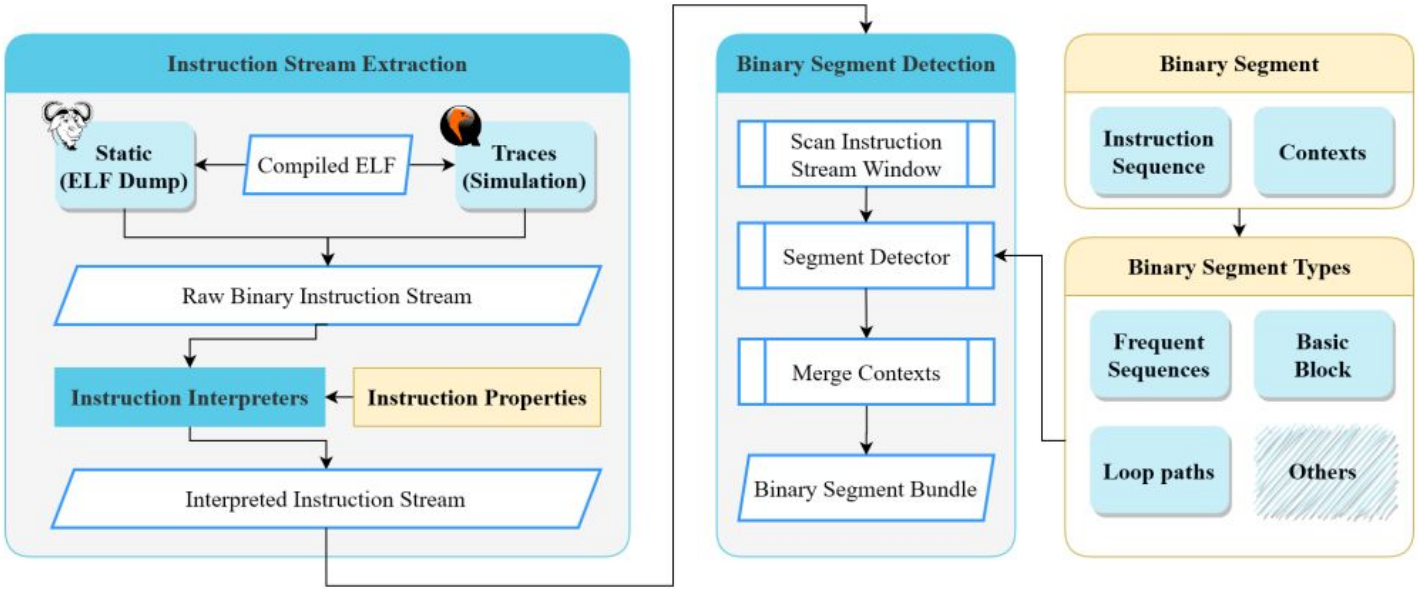
- Implemented in Java
- Starts by analysis of ELF file, or trace dump
- Produces CDFGs of repetitive patterns
- Lots still to do!



Decoding and Detection

Phase 1 - Decoding

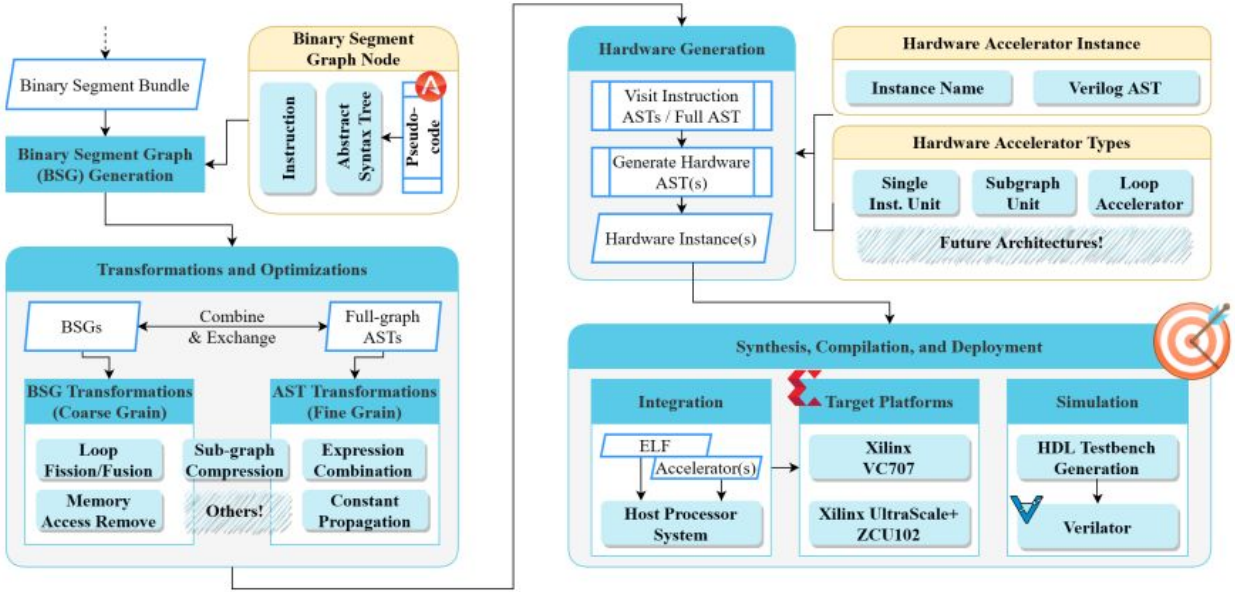
Phase 2 - Detection



Graphs, ANTLR, and Output

Phase 3 - IRs

Phase 4 - Output Generation



Simple Example: Graph Detection from Trace

Trace analysis

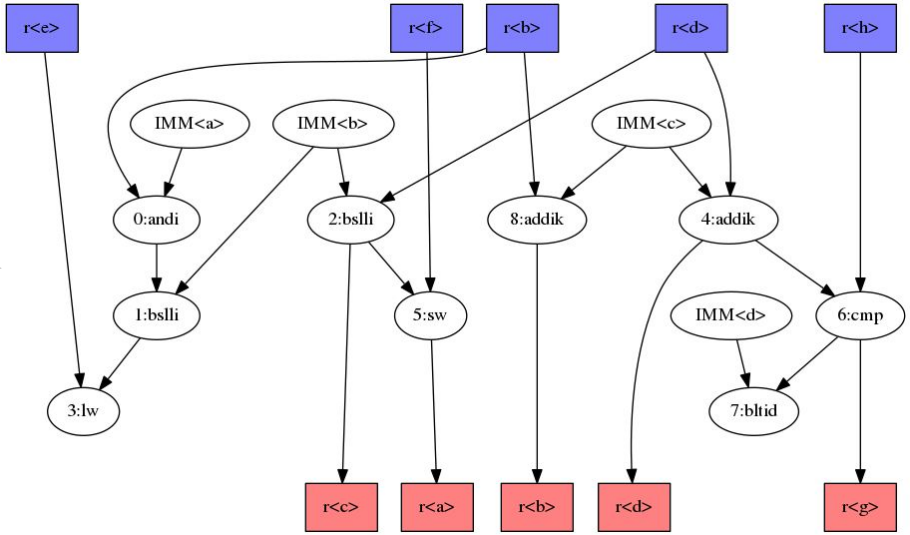
One detected loop:

Segment:

```
andi r<a>, r<b>, IMM<a>
bslli r<a>, r<a>, IMM<b>
bslli r<c>, r<d>, IMM<b>
lw r<a>, r<a>, r<e>
addik r<d>, r<d>, IMM<c>
sw r<a>, r<c>, r<f>
cmp r<g>, r<h>, r<d>
bltid r<g>, IMM<d>
addik r<b>, r<b>, IMM<c>
```

Segment Contexts

Start Addresses: 0x2994
Number of ocurrences: 33



- On-going: transforming graphs like these into hardware modules

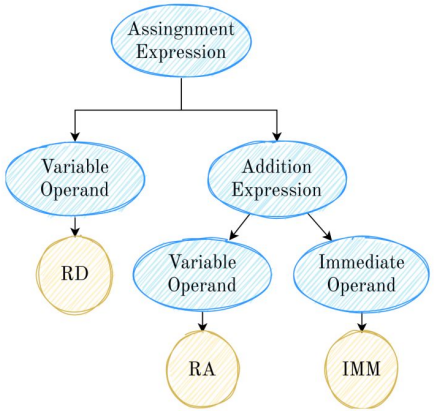
Simple Example: One MB Instruction to Verilog

Example Pseudo-code
Expresses arithmetic of instructions, using ASM fields as operands.

E.g., MicroBlaze instruction:
0x20c065e8 is
"addi r6, r0, 26088"

Pseudo-code *is*
"RD = RA + IMM;"

Generate parse tree,transform to Instruction AST, and apply execution information




```
(Simple) Example Output Code for
Single Instruction Unit
Generator
/*
* Copyright 2020 SPeCS.  * (...)
more copyright text (...)
*/
module addi_20c065e8;
output [31 : 0] r6;
input [31 : 0] r0;

// implementation for
// instruction:
// addi    r6,  r0,  0x65e8
assign r6 = r0 + 32'd26088;
endmodule
```

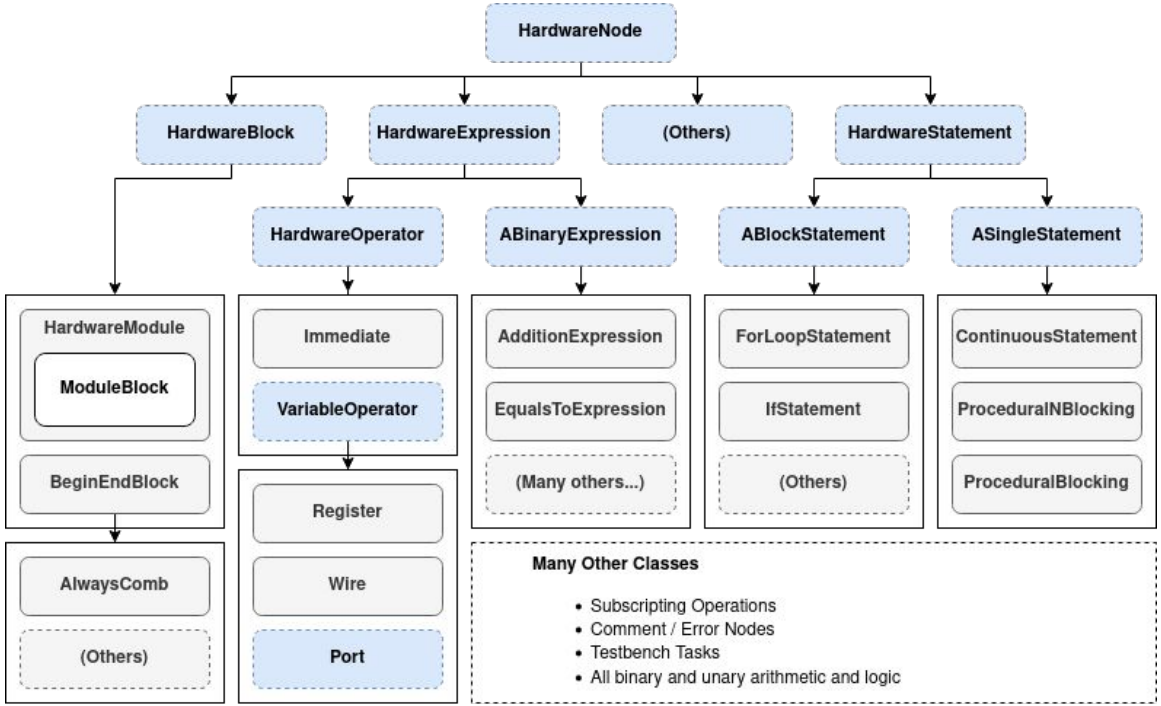
5.CrispyHDL

4. CrispyHDL

- We needed a programmatic way to generate Verilog efficiently
  - Borrow techniques from the compiler domain
  - A (nearly) complete **Verilog AST package** integrated into the Binary Translation Framework
- Verilog AST package grew → **Separate CrispyHDL project**
- **CrispyHDL**
  - Internal Java DSL for Verilog (Inspired by SpinalHDL, Chisel3, etc)
  - Generation of hardware via reusable blocks exploiting high level abstractions
    - Inheritance
    - Generics/Templates
    - Instantiation loops
    - Interfaces

Verilog Abstract Syntax Tree (AST) - Java Classes

- Each node
  - Is a Verilog element
  - Emits its respective Verilog source
- Trees of nodes are constructed via Java DSL to generate complete Verilog modules



U.PORTO FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

INESCPORTO INSTITUTO DE ENGENHARIA DE SISTEMAS E COMPUTADORES DO PORTO

Verilog Abstract Syntax Tree (AST) - Example

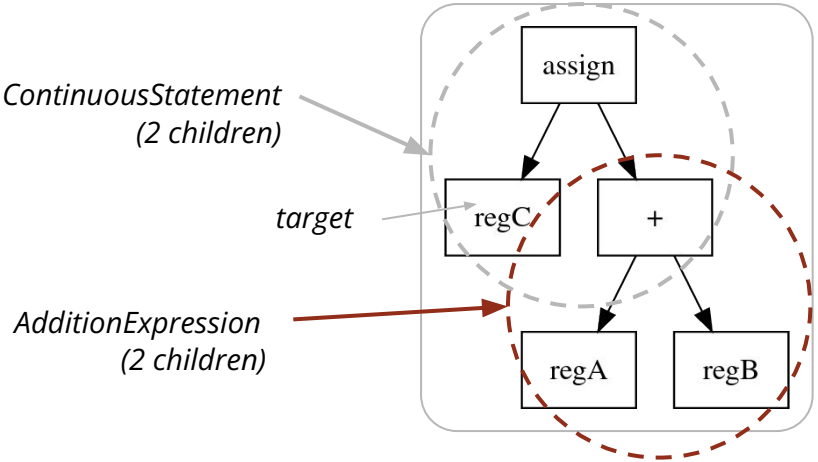
- Simple statement example
- Crispy classes are **not** meant to be explicitly instantiated like this
- The DSL (wrappers) hides this verbosity

```
var a = (new RegisterDeclaration("regA", 8)).getReference();
var b = (new RegisterDeclaration("regB", 8)).getReference();
var c = (new RegisterDeclaration("regC", 8)).getReference();

var r = new ContinuousStatement(c, new AdditionExpression(a, b));
r.emit();
```

emit

```
assign regC = regA + regB;
```

Internal tree structure



(Some) Crispy API Syntax

```java
@Test
public void workshopExample4() {

    // create a class which inherits the
    // syntax (wrapper methods)
    var ex = new HardwareModule("example");

    // creates a "WireDeclaration", but returns
    // a "Wire", which is a reference to the
    // declared name (same for Registers and Ports)
    var wire = ex.addWire("ex1", 8);

    // new port
    var a = ex.addInputPort("pA", 8);

    // create an assign at the level of the module body
    ex.assign(a, wire.lsl(2));

    // emit to stdout (eventually, to files)
    ex.emit();
}
```

- There is also syntax for
  - *if*
  - *if-else*
  - *always ff*
  - *always comb*
  - *initial*
  - etc…
- Some (early) handling of
  - Sanity checks
  - Automatic wire generation

```java
module example(pA);

    // Declarations block: Ports
    input wire [7 : 0] pA;

    // Declarations block: Wires
    wire [7 : 0] ex1;

    assign pA = ex1 << 8'd2;
endmodule //example
```

Programmatic Module Generation

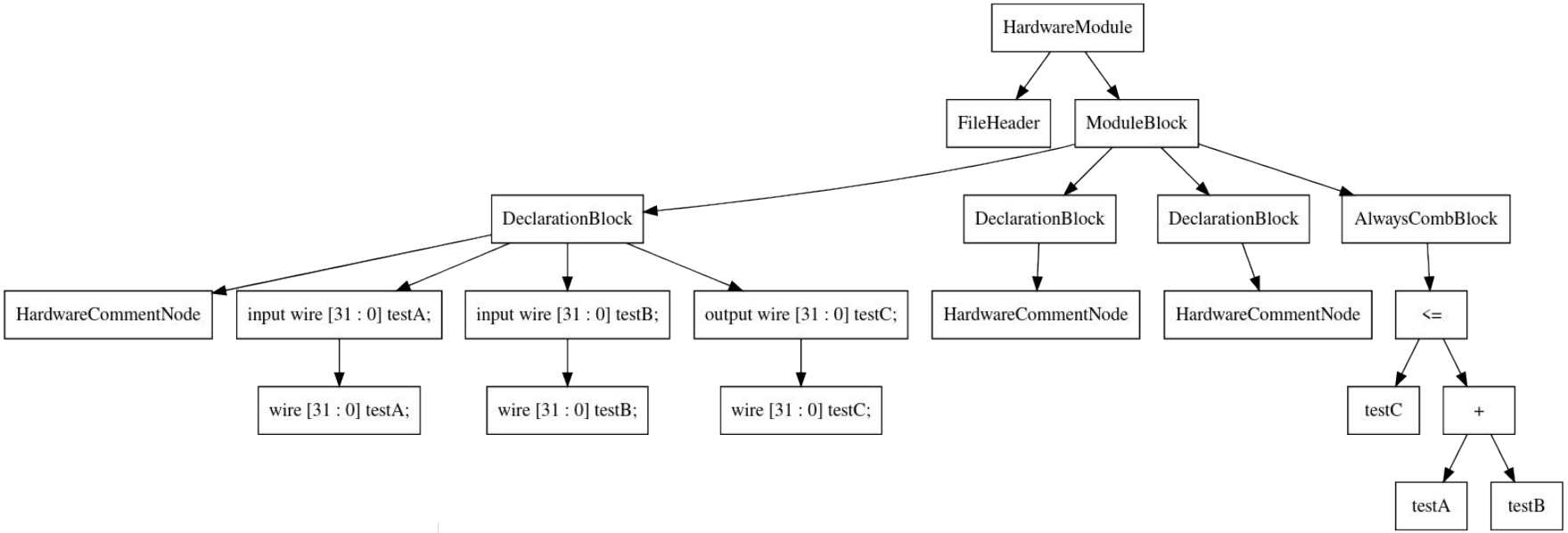
```
public void workshopExample2() {
    var adder = new HardwareModule("testAdder");
    var a = adder.addInputPort("testA", 32);
    var b = adder.addInputPort("testB", 32);
    var c = adder.addOutputPort("testC", 32);
    adder.alwayscomb()._do(c.nonBlocking(a.add(b)));

    adder.emit();
}
```

```
module testAdder(testA, testB, testC);

    // Declarations block: Ports
    input wire [31 : 0] testA;
    input wire [31 : 0] testB;
    output wire [31 : 0] testC;

    always_comb begin : comb_0
        testC <= testA + testB;
    end
endmodule //testAdder
```

- Declaring a generic module, and then adding blocks, instances, and statements to it
  - Allows for **arbitrary** module generation integrated into other flows
  - But not clear if this capability is good or bad, in terms of language use/design…

Programmatic Module Generation

The tree structure of the previous example



Explicit Module Generation via Extension

```
public class Mux2to1 extends HardwareModule {

    public InputPort i0;
    public InputPort i1;
    public InputPort sel;
    public OutputPort out;

    public Mux2to1(int bitwidth) {
        super(Mux2to1.class.getSimpleName());

        i0 = addInputPort("i0", bitwidth);
        i1 = addInputPort("i1", bitwidth);
        sel = addInputPort("sel", 1);
        out = addOutputPort("out", bitwidth);

        alwayscomb("muxBlock")._ifelse(sel.not())
                .then()._do(out.nonBlocking(i0))
                .orElse()._do(out.nonBlocking(i1));
    }
}
```

Inherits all sugar and sanity checking methods (i.e., “defines” the syntax within this class

Public members allow easier syntactic access to ports

Some repetition is required when ports depend on constructor arguments… *how to avoid?*

- This makes Crispy more similar to Chisel or SpinalHDL, but is it the best way?

Module Instantiation

- Still needs a significant amount of work!

- Difficult to:
  - Keep track of instances
  - Define the proper abstractions

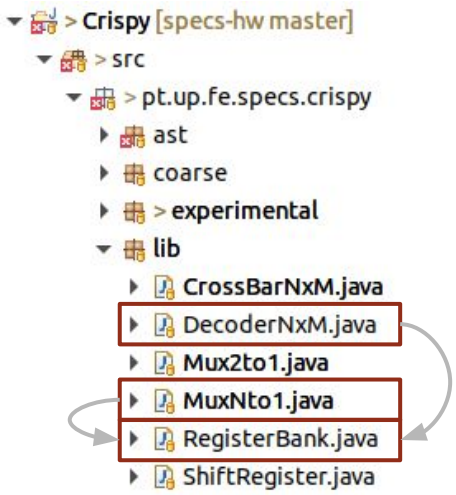
```java
public class Add extends HardwareModule {

    public InputPort inA = addInputPort("inA", 8);
    public InputPort inB = addInputPort("inB", 8);
    public OutputPort outC = addOutputPort("outC", 8);

    public Add() {
        super(Add.class.getSimpleName());
        this._do(outC.nonBlocking(inA.add(inB)));
    }
}
```

```java
public class Add3 extends HardwareModule {

    public InputPort inA = addInputPort("inA", 8);
    public InputPort inB = addInputPort("inB", 8);
    public InputPort inC = addInputPort("inC", 8);
    public OutputPort outD = addOutputPort("outD", 8);

    public Add3() {
        super(Add3.class.getSimpleName());

        var aux1 = addWire("aux1", 8);
        instantiate(new Add(), inA, inB, aux1);
        instantiate(new Add(), aux1, inC, outD);
    }
}
```

```verilog
module Add3(inA, inB, inC, outD);

    // Declarations block: Ports
    input wire [7 : 0] inA;
    input wire [7 : 0] inB;
    input wire [7 : 0] inC;
    output wire [7 : 0] outD;

    // Declarations block: Wires
    wire [7 : 0] aux1;

    Add Add_1926 (
     .inA(inA),
     .inB(inB),
     .outC(aux1)
    );

    Add Add_1555 (
     .inA(aux1),
     .inB(inC),
     .outC(outD)
    );

endmodule //Add3
```

Future library of building blocks (?)

- Writing a register bank of arbitrary bit-width and size
  - +/- 5 minutes
  - Validated manually in Vivado simulation
- Future library blocks
  - AXI interfaces?
  - Buses?
  - Caches?
  - Floating point units?



Current Application

- Master's Thesis
  - *Generating Hardware Modules via Binary Translation of RISC-V Binaries*
    - Translation of RISC-V instruction sequences into Verilog (via BTF + CrispyHDL)

```
public enum RiscvPseudocode implements InstructionPseudocode {

    add("RD = RA + RB;"),
    sub("RD = RA - RB;"),
    slt("if(signed(RA) < signed(RB)) RD = 1; else RD = 0;"),
    sltu("if(unsigned(RA) < unsigned(RB)) RD = 1; else RD = 0;"),
    etc...
```

→ *Programmatically Generated Modules*

- Reimplementing the Loop Accelerator (from IEEE TLVSI 2019 paper)
  - Easier/faster generation of architecture parameters
  - Integrated with loop extraction and modulo-scheduling
  - Future (partially implemented) integration with synthesis tools, reports, etc
    - e.g. via generation of TCL scripts for Vivado

Future Direction for CrispyHDL?

- Better abstraction and syntax
  - Variable names via reflection?
  - Better state keeping for module instantiation?
- Base for CGRA Architecture Exploration
  - Design space exploration of CGRA variations
  - Joint software / hardware compilation
- External DSL
  - Tentative name: *CrunchyDSL*
  - A dedicated parser for Crunchy to translate to internal Crispy nodes
  - Avoids limitations of having Crispy implemented over Java
  - Allows for context specific rules for the language

End!

Q&A?