Fourth International Workshop on

FPGAs for Software Programmers (FSP 2017)

September 7, 2017, Ghent, Belgium

co-located with

International Conference on Field Programmable Logic and Applications (FPL)

FPGA-based Acceleration: we need source to source compilers!

João M.P. Cardoso

João Bispo, Pedro Pinto, Luís Reis, Tiago Carvalho, Ricardo Nobre, and Nuno Paulino

University of Porto, FEUP/INESC-TEC, Porto, Portugal

Email: jmpc@acm.org

U.PORTO

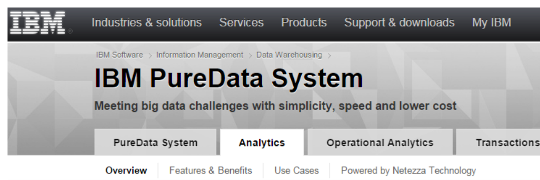
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

INESCTEC TECNOLOGIA E CIÊNCIA | LABORATÓRIO ASSOCIADO

SPeCS Special-Purpose Computing Systems, languages and tools

An Exciting Reconfigurable Computing Era!

Widely spreading…



PureData System for Analytics

"The FPGAs are 40 times faster than a CPU at processing Bing's custom algorithms, Burger says."

A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services

Andrew Putnam Adrian M. Caulfield Eric S. Chung Derek Chiou[1]
Kypros Constantinides[2] John Demme[3] Hadi Esmaeilzadeh[4] Jeremy Fowers
Gopi Prashanth Gopal Jan Gray Michael Haselman Scott Hauck[5] Stephen Heil
Amir Hormati[6] Joo-Young Kim Sitaram Lanka James Larus[7] Eric Peterson
Simon Pope Aaron Smith Jason Thong Phillip Yi Xiao Doug Burger

Microsoft

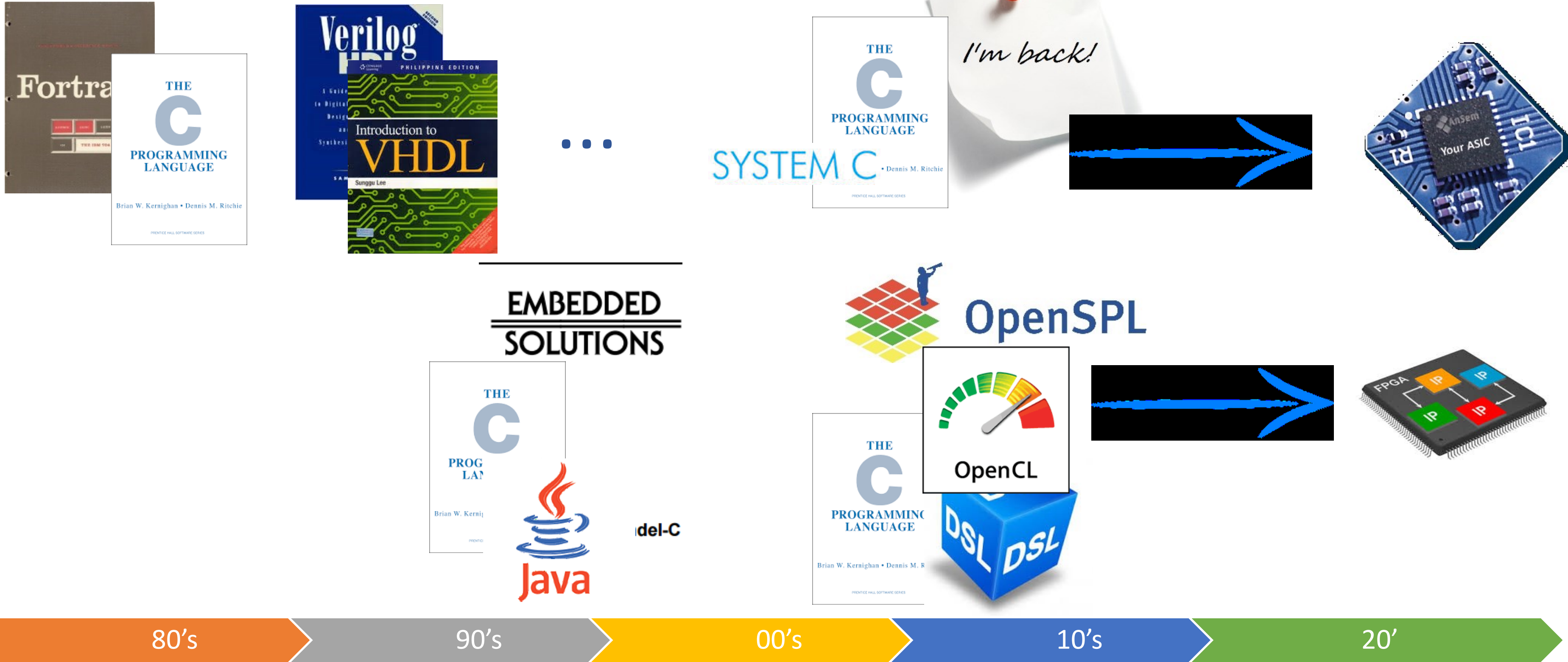
FPGA Implementation of GZIP Compression and Decompression for IDC Services

Jian Ouyang, Hong Luo, Zilong Wang, Jiazi Tian, Chenghui Liu and Kehua Sheng
System Group, Baidu Inc.
ouyangjian@baidu.com, tianjiazi@baidu.com, liuchenghui@baidu.com, shengkehua@baidu.com
Electronic Engineering Department, Tsinghua University
Beijing, China
hongluo@tsinghua.edu.cn, zilongwang@mails.tsinghua.edu.cn

Active SSD Design for Energy-efficiency Improvement of Web-scale Data Analysis

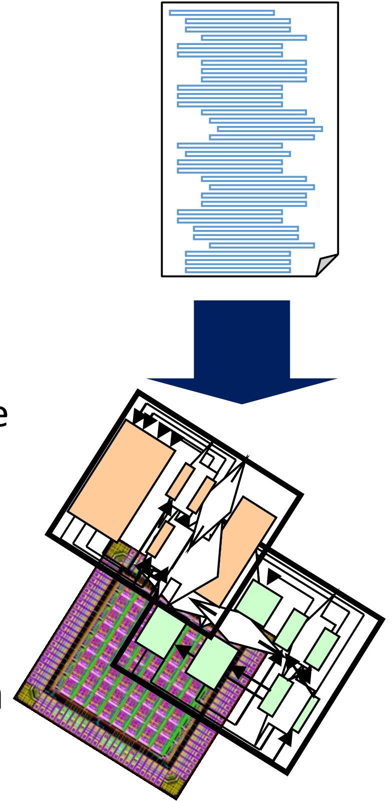
Jian Ouyang[1], Shiding Lin[1], Zhenyu Hou[1], Peng Wang[2], Yong Wang[1], Guangyu Sun[2],
[1]Baidu, Inc.
[2]Center for Energy-efficient Computing and Applications, Peking University
{ouyangjian, linshiding, houzhenyu, wangyong03}@baidu.com, {wang_peng, gsun}@pku.edu.cn

Compiling to hardware: Timeline



Compilation to FPGAs (hardware)

- From software to hardware
  - Generating hardware specific to the input software
  - Achieving performance benefits (acceleration), energy savings,…
- Of paramount importance to the mainstream adoption of FPGAs
  - Efficient compilation will improve designer productivity and will make the use of FPGA technology viable for **software programmers**
- The Challenge:
  - Added complexity of the extensive set of execution models supported by FPGAs makes efficient compilation (and programming) very hard
  - We have not yet solved the parallel programming problem, sort of…
- High-Level Synthesis (hardware generation from C) has become a real solution!

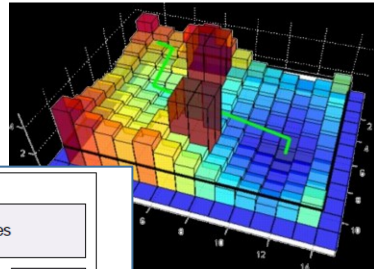


Outline

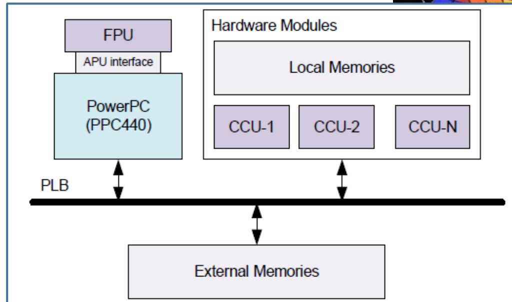
- Intro
- Why source to source compilers?
- Simple code restructuring example
- Our source to source compilation approaches
- Our source to source compilers
- Ongoing work
- Some challenges
- Conclusion

Why source to source compilers?

Code Restructuring: 3D Path Planner

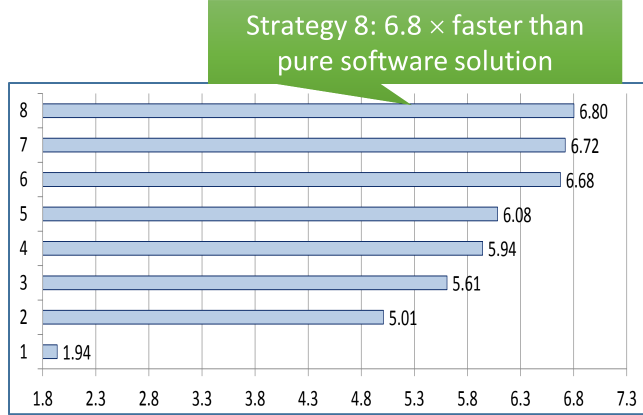


- Target: ML507 Xilinx Virtex-5 board, PowerPC@400 MHz, CCUs@100 MHz



Optimization	Strategy 1	2	3	4	5	6	7	8
Loop fission and move		✓	✓	✓	✓	✓	✓	✓
Replicate array 3×					✓	✓	✓	✓
Map gridit to HW core	✓	✓	✓	✓	✓	✓	✓	✓
Pointer-based accesses and strength reduction			✓	✓	✓	✓	✓	✓
Unroll 2×	✓	✓	✓	✓	✓	✓	✓	✓
Eliminating array accesses	✓	✓	✓	✓	✓	✓	✓	✓
Move data access								✓
Specialization → 3 HW cores							✓	✓
Transfer pot data according to gridit call				✓		✓	✓	✓
Transfer obstacles data according to gridit call			✓	✓	✓	✓	✓	✓
On-demand c								

FPGA resources	Implementation 1	2,3,4	5,6	7,8
# Slice Registers as FF	901	939	956	2,470
# Slice LUTs	1,182	1,284	1,308	2,148
# occupied Slices	531	663	642	1,004
# BlockRAM/# DSP48Es	34/6	34/6	98/6	98/12



See: J. M. P. Cardoso, et al., **Specifying Compiler Strategies for FPGA-based Systems.** FCCM 2012

Source: EU-Funded FP7 REFLECT project

Example of a strategy from the ANTAREX project:

- Create multiple versions of function “A”
- Insert calls to timers for measuring the execution time of the function
- Substitute the call to the original function with the possibility to execute one of the versions based on a parameter
- Instantiate an autotuner and insert calls to the autotuner and communication of execution time
- Use the parameter output by the autotuner to select between the versions of the function at runtime
- Apply to each version a different optimization strategy

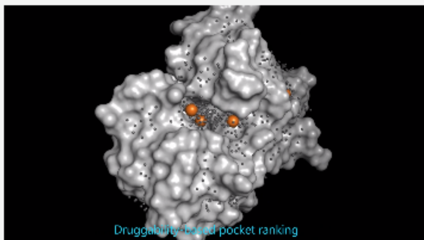
All these steps are performed at the source code level!

All these steps can be specified as (LARA) recipes automatically applied to source code!

Silvano et al., ACM CF’2016

Experiments make evident the importance of source to source transformations

Accelerating Personalized Drug Discovery: A Performance Exploration

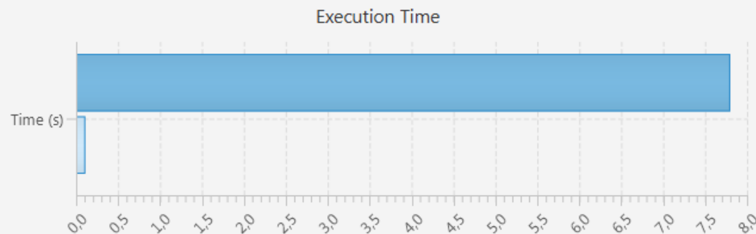


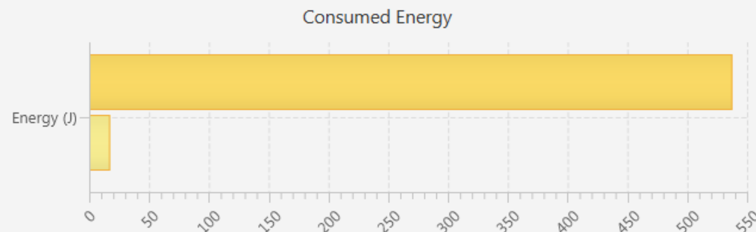
Current drug discovery focus is 'one drug fits all'. We need a shift towards personalized medicine: design drugs based on specific group response rather than specific disease.

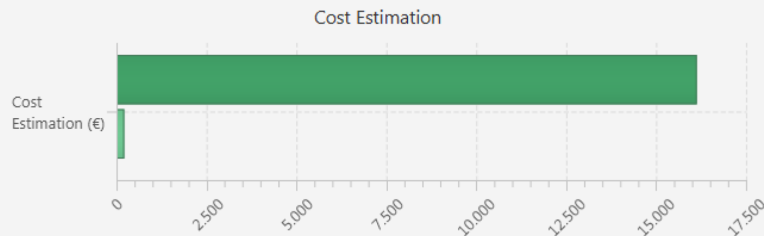
Performance exploration of an application, based on the LiGen workflow (Dompé & CINECA), that performs rotations on ligands in order to find the best dockings on protein pockets. Time and energy consumption measurements were performed for a single HPC node.

O3 + O4
O1 + O2 + O3
O1 + O3 + O4
O3 + O4 + O5
O3 + O4 + O5 + O6
O3 + O4 + O5 + O6 + O7
O3 + O4 + O5 + O6 + O7 + O8

O1: float to double in innerloop of Measureoverlap to avoid ca
O2: parallelize matchProbesShape
O3: AoS to SoA
O4: parallelize calls to matchProbeShape
O5: double to float in MeasureOverlap and Distance?







Target: 2 × Intel Xeon CPU E5-2630 v3 @ 2.40GHz (8-core CPUs)

Why source to source compilers?

- Translate from one programming language to another programming language
- Take advantage of mature tool flows
  - backend, target-aware, compilers, synthesis tools
- Apply target-aware and/or tool flow-aware code transformations

Lang. A

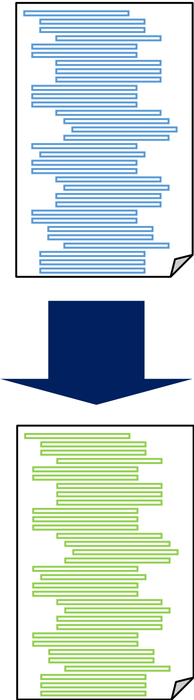
Lang. B

Lang. A

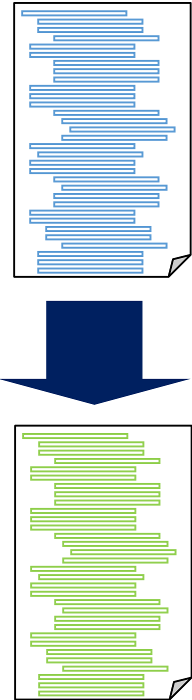
Lang. A

Source to source compilation

- Code optimizations (loop unrolling, loop tiling, etc.)
- Task-level parallelism and pipelining
- Generation of multiple code versions (multiversioning)
- Specialization/customization according to data
- Memoization
- Hardware/software partitioning (including insertion of synchronization and communication primitives)
- Instrumentation
- …



Source to source compilation



- Target code is legible (good for debugging)!
- Not tied to a specific target compiler (tool flow) or target Architecture!
- Not all optimizations can be done at source code level!
- Some code transformations are too specific and without enough application potential to justify inclusion in a compiler (unless the application is too important and must be continuously reshaped)

Code restructuring

- Manual
  - Programmers need to know the impact of code styles and structures on the generated architecture (similar to the HDL developers, although in a different level)
- Fully automatic with a source to source compiler (refactoring tool)
  - Need to devise the code transformations to apply and their ordering
  - Need source to source compilers integrating a vast portfolio of code transformations!
- Semi-automatic with a source to source compiler (refactoring tool)
  - Code transformations automatically applied but guided by users
  - Users can define their own code transformations!

Simple code restructuring example

Code Restructuring: FIR Example

```
// x is an input array
// y is an output array
#define c0 2, c1 4, c2 4, c3 2
#define M 256 // no. of samples
#define N 4 // no. of coeff.
int c[N] = {c0, c1, c2, c3};
...
// Loop 1:
for(int j=N-1; j<M; j++) {
  output=0;
  // Loop 2:
  for(int i=0; i<N; i++) {
    output+=c[i]*x[j-i];
  }
  y[j] = output;
}
```

1

II=2

```
// Loop 1
for(int j=3; j<M; j++) {
  x_3=x[j];
  x_2=x[j-1];
  x_1=x[j-2];
  x_0=x[j-3];
  output=c0*x_3;
  output+=c1*x_2;
  output+=c2*x_1;
  output+=c3*x_0;
  y[j] = output;
}
```

1 sample per 2 clock cycles

2

II=1

```
x_0=x[0];
x_1=x[1];
x_2=x[2];
// Loop 1
for(int j=3; j<M; j++) {
  x_3=x[j];
  output=c0*x_3;
  output+=c1*x_2;
  output+=c2*x_1;
  output+=c3*x_0;
  x_0=x_1;
  x_1=x_2;
  x_2=x_3;
  y[j] = output;
}
```

1 sample per clock cycle

Code Restructuring: FIR Example

1

II=2

```
// Loop 1
for(int j=3; j<M; j++) {
    x_3=x[j];
    x_2=x[j-1];
    x_1=x[j-2];
    x_0=x[j-3];
    output=c0*x_3;
    output+=c1*x_2;
    output+=c2*x_1;
    output+=c3*x_0;
    y[j] = output;
}
```

1 sample per 2 clock cycles

2

II=1

```
x_0=x[0];
x_1=x[1];
x_2=x[2];
// Loop 1
for(int j=3; j<M; j++) {
    x_3=x[j];
    output=c0*x_3;
    output+=c1*x_2;
    output+=c2*x_1;
    output+=c3*x_0;
    x_0=x_1;
    x_1=x_2;
    x_2=x_3;
    y[j] = output;
}
```

1 sample per clock cycle

3

II=1

```
// Loop 1
for(int j=3; j<M; j+=2) {
    x_3=x[j];
    output=c0*x_3;
    output+=c1*x_2;
    output+=c2*x_1;
    output+=c3*x_0;
    x_0=x_1;
    x_1=x_2;
    x_2=x_3;
    y[j] = output;
    x_3=x[j+1];
    output=c0*x_3;
    output+=c1*x_2;
    output+=c2*x_1;
    output+=c3*x_0;
    x_0=x_1;
    x_1=x_2;
    x_2=x_3;
    y[j+1] = output;
}
```

2 samples per clock cycle

See: João M. P. Cardoso, Markus Weinhardt, **High-Level Synthesis.** FPGAs for Software Programmers 2016.

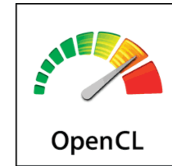
Our source to source compilation approaches

Assumptions considering HLS from C

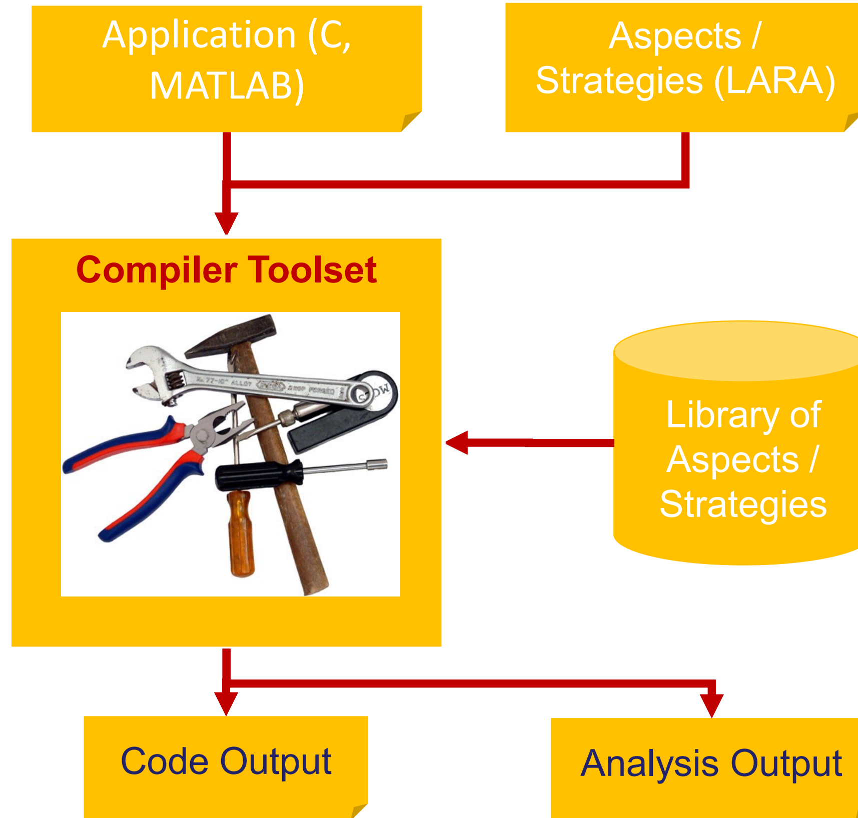
- It is possible to generate efficient hardware accelerators from "massaged" C code (+ directives)
- Directives will aid compilers with the information they cannot automatically extract/expose
- Directives will instruct compilers to apply what they cannot easily devise
- HLS will be extended to deal with directive driven programming models (e.g., OpenMP) + concurrency

Focus

- C/OpenCL (+ directives + directive driven programming models) as our intermediate representation (IR)
  - Compiler generates target-specific code in this IR
  - Then, HLS and backend FPGA tools are used
- This IR still misses other ways to express coarse-grained concurrency (e.g., communicating sequential processes, OpenMP directives)



LARA-based tool flow



Aspect-Oriented

Application
(C, MATLAB)

Aspects and Strategies
(LARA)

LARA Front-End

Design-Space Exploration (DSE)

Source to Source

Aspect-IR

hardware/software partitioning
code insertion

Best Practices

Kernels for Hw/Sw Components (C) + Annotations

Hardware/Software Cores

Compiler Toolset

C Front-End

high-level optimizations:
-function inlining
-loop unrolling
-loop tilling

CDFG-IR

Optimizer
(Software/Hardware)

Hardware/Softwar

middle-level optimizations:
- word length analysis

CDFG-IR

weaving

Back-End (code generators)

low-level optimizations
retargetability

Assembly

VHDL-RTL

Compiler Toolset

Source: EU-Funded FP7 REFLECT project

LARA-based tool flow

Application

```
void filter_subband(float
z[512], float s[32], float
m[32][64]) {
  ...
  for (i=0;i<32;i++) {
    s[i]= 0;
    for (j=0;j<64;j++) {
      s[i] += m[i][j] * y[j];
    }
  }
  ...
```

Compiler Toolset

Aspects and Strategies

```
aspectdef monitor1
  select function{}.var{“s”} end
  apply
    insert.after %{if([[$var.usage]] >= 10)
      printf(“Warning: value >= 10!\n”);} %
  end
  condition $var.is_write end
end
```

Program elements

Advices (actions)

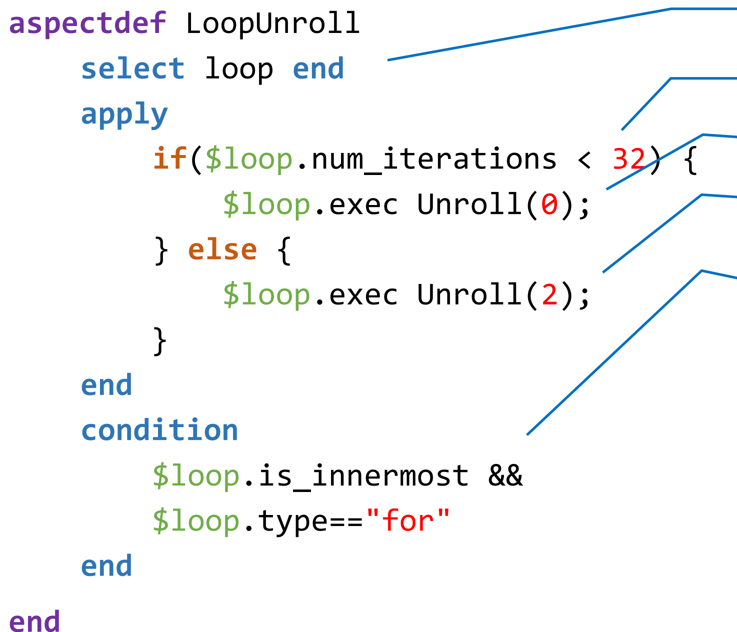
Condition

Code Output

```
...
 for (i=0;i<32;i++) {
   s[i]= 0;
   if(s[i] >= 10)  printf(“Warning: value >= 10!\n”);
   for (j=0;j<64;j++) {
     s[i] += m[i][j] * y[j];
     if(s[i] >= 10)  printf(“Warning: value >= 10!\n”);
   }
 }
...
```

LARA Action: Code Instrumentation

LARA strategies: simple loop unrolling example



```
aspectdef LoopUnroll
    select loop end
    apply
        if($loop.num_iterations < 32) {
            $loop.exec Unroll(0);
        } else {
            $loop.exec Unroll(2);
        }
    end
    condition
        $loop.is_innermost &&
        $loop.type=="for"
    end
end
```

- More sophisticated analyses/strategies are possible:
  - Using attributes
  - JavaScript code

LARA strategies

- Recursively unroll loops fully or 2×, depending on their chacteristics

Input Program

```
...
for (i=0;i<64;i++) {
  y[i] = 0;
  for (j=0;j<8;j++)
    y[i] += z[i+64*j];
}
for (i=0;i<32;i++) {
  s[i] = 0;
  for (j=0;j<64;j++)
    s[i] += m[i*32 +j] * y[j];
}
...
```

```
...
for (i=0;i<64;i++) {
  y1 = z[i]; ...
  y1 += z[i+64*7];
  y[i] = y1;
  y1 = z[i+1]; ...
  y1 += z[i+1+64*7];
  y[i+1] = y1;
}
for (i=0;i<32;i+=2) {
  s1 = 0;
  for (j=0;j<64;j+=2) {
    s1 += m[i*32+j]*y[j];
    s1 += m[i*32+j+1]*y[j+1];
  }
  s[i] = s1;
  ...
}
...
```

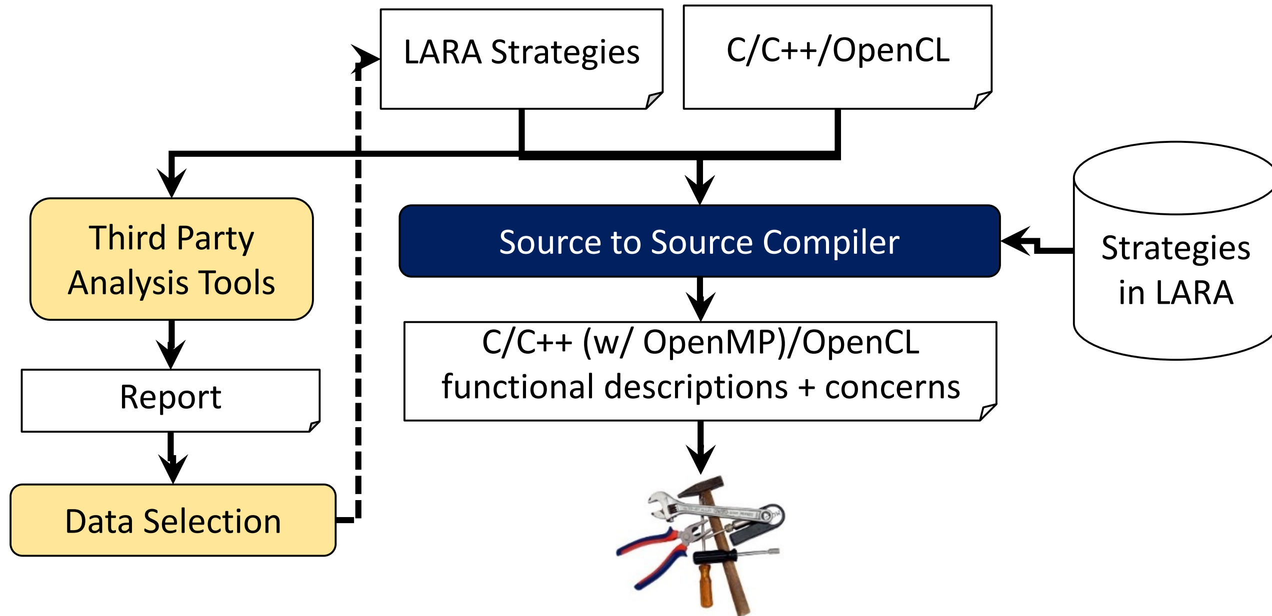
Strategies

```
aspectdef Strategy
  input fn="f1" end
  select function{name==fn} end
  apply
    do {                    } while($function.changed);
  end
end
```

```
aspectdef  loopunroll
 input  niter1=10, niter2=20 end
 select  loop{type=="for"}  end
 apply
   exec loopscalar;
   if($loop.num_iter <= niter1) {
      exec loopunroll(k:"full");
   } else if($loop.num_iter <= niter2) {
       exec loopunroll(k:2); $loop.already="true";
   }
 end
 condition
   !$loop.already && $loop.is_innermost &&
   $loop.numIterIsConstant;
 end
end
```

LARA-based tool flow

- Multistage approach



Our source to source compilers

Our source to source compilers

- The MATISSE MATLAB Compiler
  - MATLAB-to-C/OpenCL
  - http://specs.fe.up.pt/tools/matisse
- MANET
  - C to C compiler (based on Cetus)
  - http://specs.fe.up.pt/tools/manet
- Clava
  - C/C++-to-C/C++ compiler (Clang as frontend)
  - http://specs.fe.up.pt/tools/clava
- Kadabra
  - Java to Java compiler (based on Spoon)
  - http://specs.fe.up.pt/tools/kadabra

Clava

Strategies (LARA)

Application Code (C/C++)

Larai

C/C++ Weaver

Clava C/C++ AST

Clang based Frontend

Application Code (C/C++, OpenMP)

- Clang based source to source compiler controlled by LARA
- Code refactoring techniques to
  - increase performance, energy efficiency
  - support/help dynamic adaptivity schemes
  - expose autotuning opportunities
- MPI/OpenMP Strategies

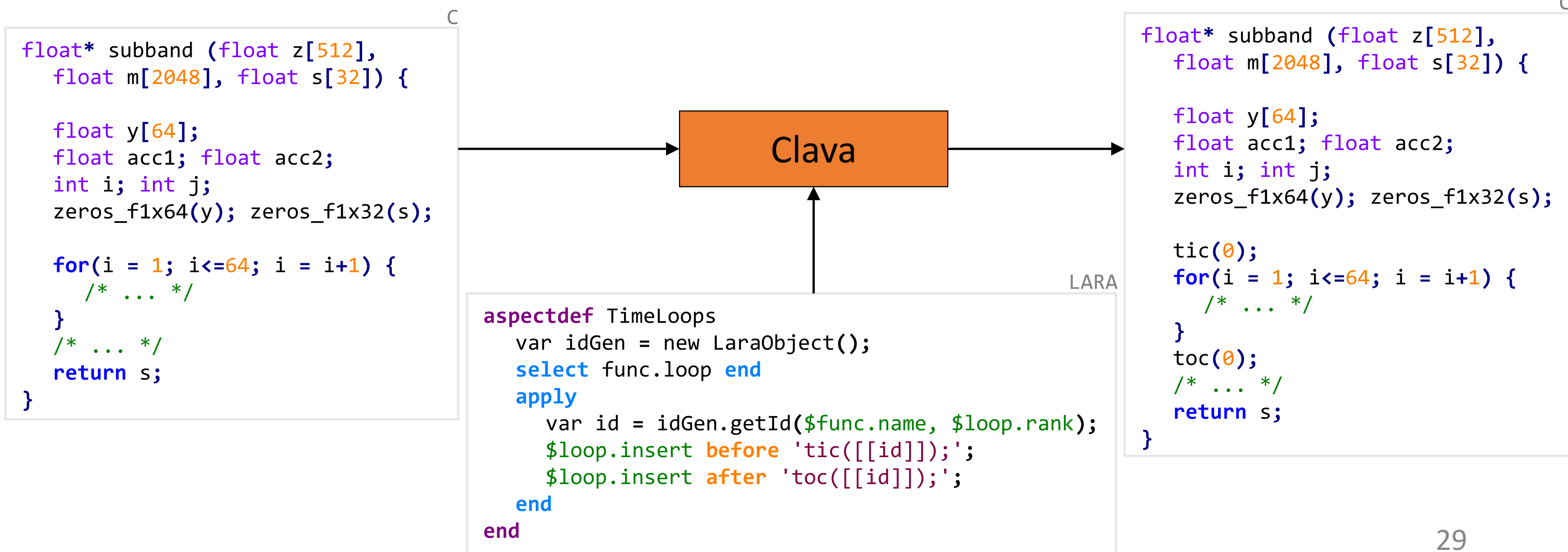
http://specs.fe.up.pt/tools/clava

http://www.antarex-project.eu

Main contact: João Bispo

Clava: Guiding Compilation and Transformations

Clava receives the C code for the *subband* function and uses an aspect which inserts code to measure the execution time of each loops in the code and print a report at the end of execution



MATISSE

- MATLAB Compiler Framework:
  - MATLAB-to-MATLAB compilation
  - MATLAB-to-C/OpenCL compilation
- Web Demo:
  - http://specs.fe.up.pt/tools/matisse

MATLAB Code
MATLAB Front-End
MATLAB IR
LARA Aspects
MWeaver
MATLAB IR + Information
Analysis & Optimizations
Optimized MATLAB IR
MATLAB Generator
Transformations (C, OpenCL and Target-Dependent)
C-oriented MATLAB IR
OpenCL-oriented MATLAB IR
MATLAB Code
C Code Generator
OpenCL Code Generator
C Code
OpenCL Code

Main contact: Luís Reis

MATISSE: Guiding Compilation and Transformations

MATISSE receives the MATLAB code for the *subband* function and a LARA aspect which defines the types (*single* precision floating point in this case) and shape of the variables used in the function

MATLAB

```
function s = subband(z, m)
   for i = 1:64
      acc1 = 0;
      for j = 0:7
       acc1 = acc1 + z(i+64*j);
      end
      y(i) = acc1;
   end
%...
```

LARA

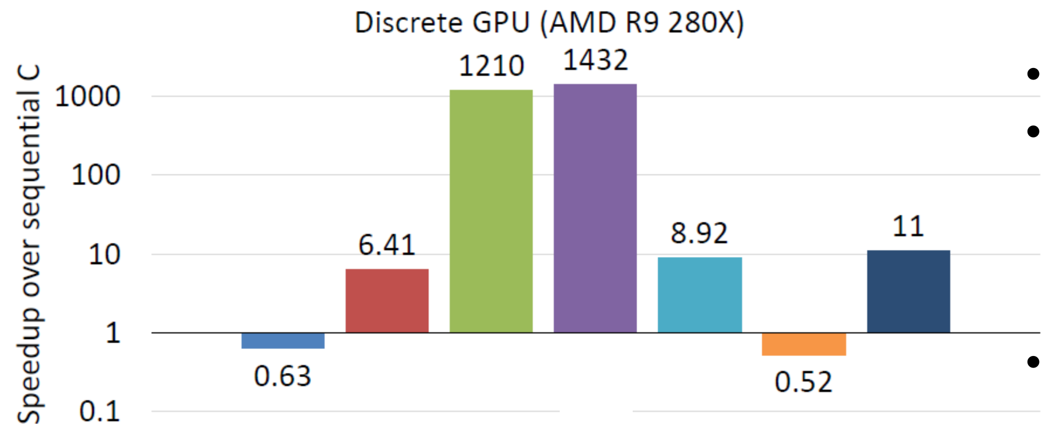
```
aspectdef DefineTypesAsSingle
   var typeDef = {
      z: 'single[1][512]',
      m: 'single[1][2048]',
      y: 'single[1][64]',
      s: 'single[1][32]'};
   call defineTypes('subband', typeDef);
end
```

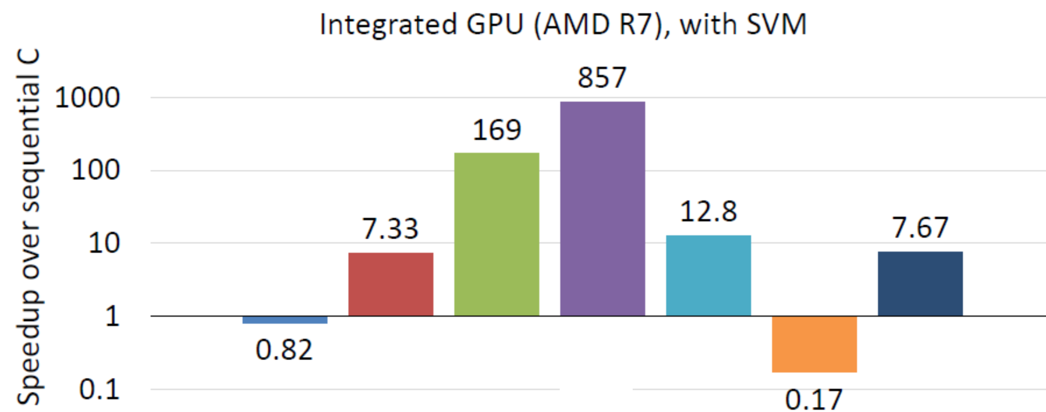
MATISSE

C

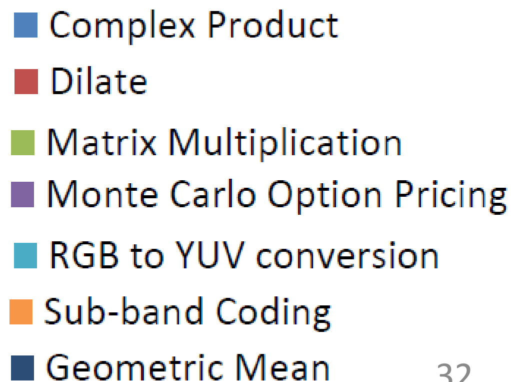
```
float* subband (float z[512],
   float m[2048], float s[32]) {

   float y[64];
   float acc1; float acc2;
   int i; int j;
   zeros_f1x64(y); zeros_f1x32(s);

   for(i = 1; i<=64; i = i+1) {
      /* ... */
   }
   /* ... */
   return s;
}
```

MATISSE C vs OpenCL (GPU target)





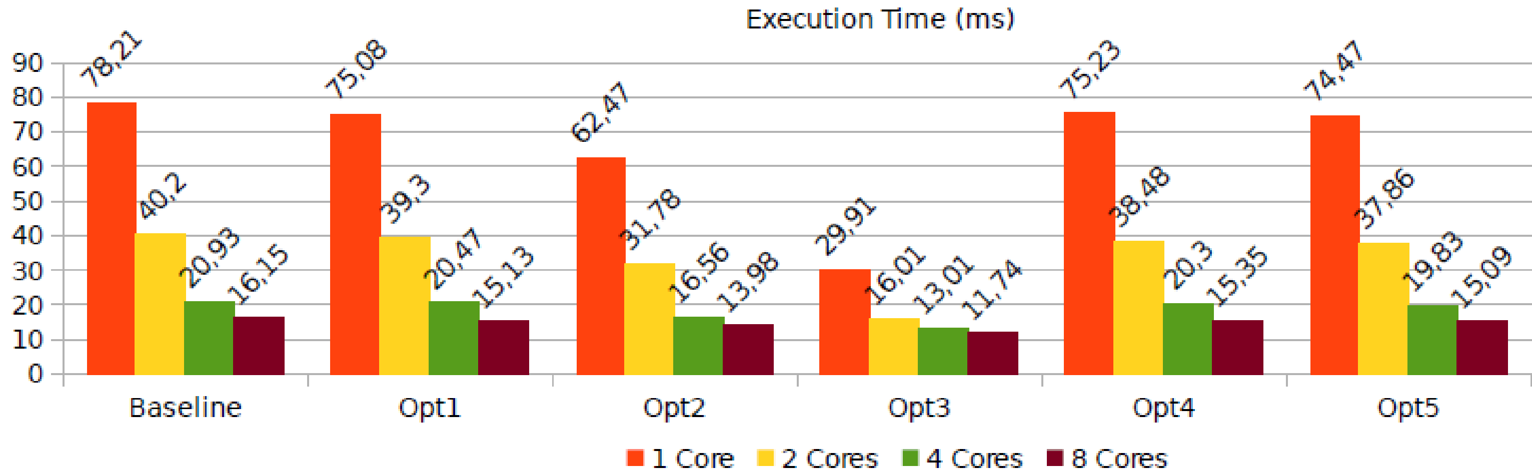
- MATISSE OpenCL vs MATISSE C
- Target: PC with an AMD A10-7850K APU (@4.10 GHz), w/ 8 GB of DDR3 RAM, a discrete Radeon R9 280X GPU and an integrated Radeon R7 GPU
- Target code compiled with GCC 4.9.2 with -O3



MATISSE OpenCL (FPGA)

Baseline MATISSE Generated Code
Opt1 *xcl_pipeline_workitems* directives
Opt2 2-Element vectorization (i.e. uchar2)
Opt3 4-Element vectorization (i.e. uchar4)
Opt4 *xcl_pipeline_loop* directives
Opt5 Opt4 + *xcl_array_partition* + unrolling hints

- Generation of OpenCL and use of Xilinx SDAccel
- Alpha Data ADM-PCIE-KU3 with a Kintex-6 XCKU060 FPGA

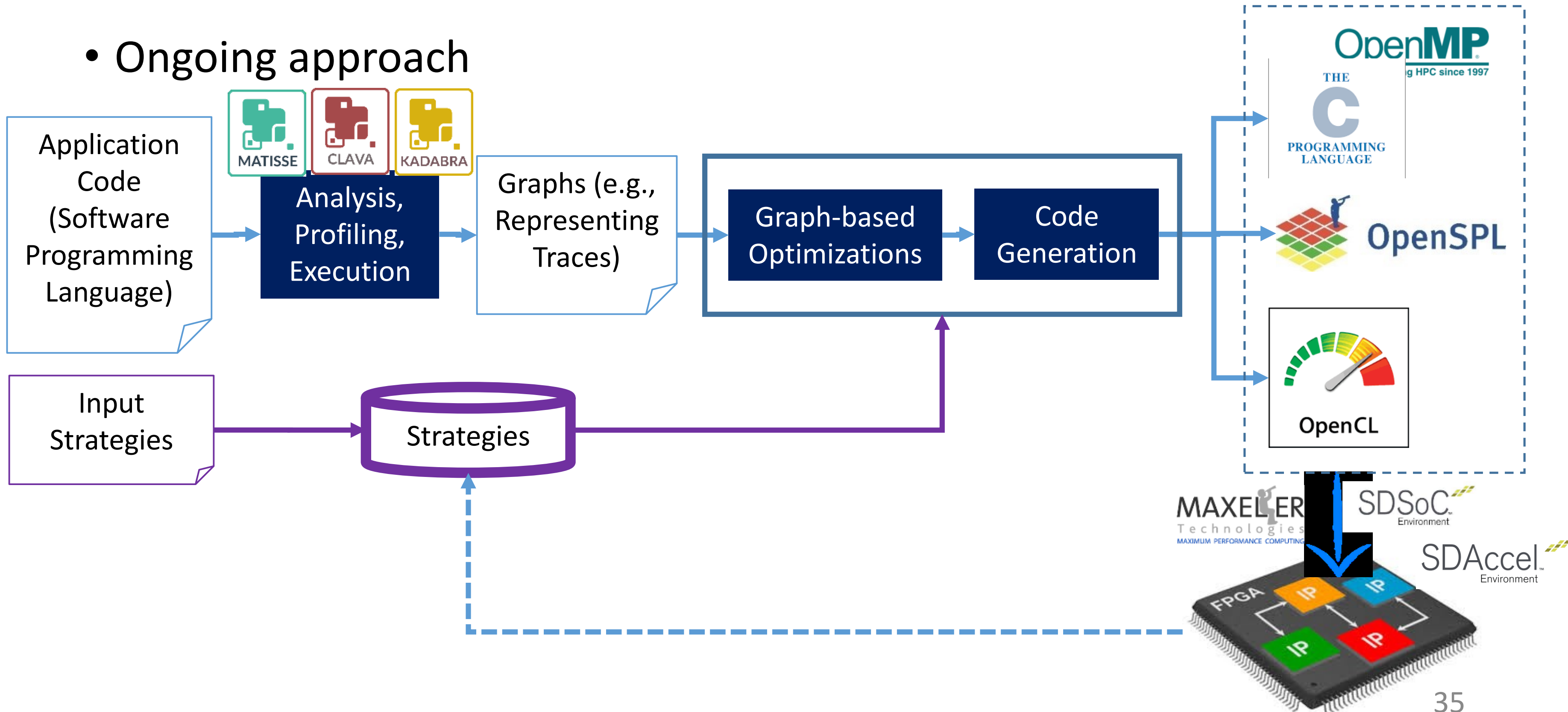


Main contact: Nuno Paulino

Ongoing work

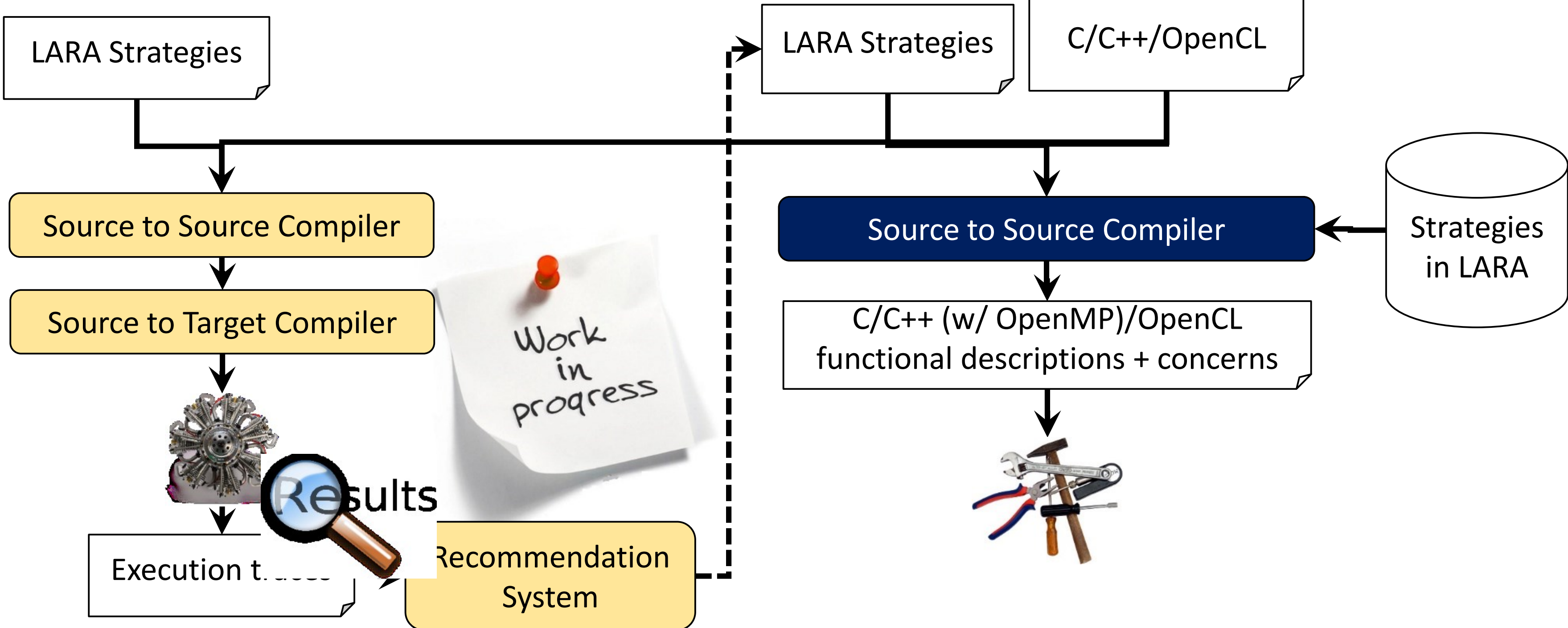
Code restructuring

- Ongoing approach



Source to source compilers

- Multistage approach (ongoing work)



Challenges

No universal programming language

Language Types (click to hide)

Language Rank	Types	Spectrum Ranking
1. C	Mobile, Enterprise, Embedded	99.2
2. C++	Mobile, Enterprise, Embedded	95.5
3. Assembly	Embedded	69.7
4. Arduino	Embedded	62.0
5. D	Enterprise, Embedded	49.7
6. Haskell	Enterprise, Embedded	44.9
7. VHDL	Embedded	41.6
8. Verilog	Embedded	32.9
9. Erlang	Embedded	31.2
10. Ada	Enterprise, Embedded	30.8
11. TCL	Enterprise, Embedded	21.1
12. Ladder Logic	Embedded	16.9
13. Forth	Embedded	2.4

Language Types (click to hide)

Language Rank	Types	Spectrum Ranking
1. Java	Web, Mobile, Enterprise	100.0
2. C	Mobile, Enterprise, Embedded	99.2
3. C++	Mobile, Enterprise, Embedded	95.5
4. Python	Web, Enterprise	93.4
5. C#	Web, Mobile, Enterprise	92.2
6. PHP	Web	84.6
7. Javascript	Web, Mobile	84.3
8. Ruby	Web	78.6
9. R	Enterprise	74.0
10. MATLAB	Enterprise	72.6
11. SQL	Enterprise	70.5
12. PERL	Web, Enterprise	70.1
13. Assembly	Embedded	69.7

Interactive: The Top Programming Languages, IEEE Spectrum's 2014 Ranking, By Stephen Cass, Nick Diakopoulos & Joshua J. Romero, http://spectrum.ieee.org/static/interactive-the-top-programming-languages

Challenges

- Software is software!
- Devise code transformations and sequence of code transformations to apply
- Deal with the high engineering efforts to automate code transformations
- Some code transformations are difficult to automate (e.g., AoS to SoA, non-streaming to streaming)

Conclusion

- Compilation to FPGAs needs to deal with two complexities:
  - Software complexity (e.g., lines of code, dynamic data structures, objects)
  - Hardware complexity (resources, more features)
- Compilation to FPGAs needs:
  - More efficient and aggressive code restructuring
  - To avoid the possible show stopper provided by the C programming language
- **Our approach: source to source compilation and HLS tools as the new backends for advanced compilation!**

Thank you! Questions?

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

INESCTEC
TECNOLOGIA E CIÊNCIA
LABORATÓRIO ASSOCIADO

SPeCS
Special-Purpose Computing
Systems, languages and tools

Acknowledgments

SMILES

CONTEXTWA

and PhD grants

Announcement:

Published: 15th June 2017
Imprint: Morgan Kaufmann

URL: https://www.elsevier.com/books/embedded-computing-for-high-performance/cardoso/978-0-12-804189-5