AUTOMATIC CODE RESTRUCTURING FOR FPGAS: CURRENT STATUS, TRENDS AND OPEN ISSUES

Special Day on “Embedded Meets Hyperscale and HPC”

João MP Cardoso
jmpc@acm.org
Compiling to hardware: Timeline

80’s | 90’s | 00’s | 10’s | 20’s
Compiling to FPGAs (hardware)

• Of paramount importance for allowing software developers to map computations to FPGA-based accelerators
  • Efficient compilation will improve designer productivity and will make the use of FPGA technology viable for software programmers

• Challenge:
  • Added complexity of the extensive set of execution models supported by FPGAs makes efficient compilation (and programming) very hard

• Years of research on High-Level Synthesis (mostly on hardware generation from C) and adoption of mature compiler frameworks are resulting in the effective use of HLS
Outline

• Intro
• Why source to source compilers?
• Code restructuring
• Some approaches for code restructuring
• Our ongoing work
• Conclusion
• Future work
Why source to source compilers?

• There are many optimizations and code transformations that can be explored at the source code level
• Target code is still legible
• Not tied to a specific target compiler (tool flow) or target Architecture!

But:
• 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 code is too important and must be regularly used/modified/extended)
Source level code transf.: 3D Path Planner

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

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

FPGA resources

<table>
<thead>
<tr>
<th>Implementation</th>
<th>1</th>
<th>2,3,4</th>
<th>5,6</th>
<th>7,8</th>
</tr>
</thead>
<tbody>
<tr>
<td># Slice Registers as FF</td>
<td>901</td>
<td>939</td>
<td>956</td>
<td>2,470</td>
</tr>
<tr>
<td># Slice LUTs</td>
<td>1,182</td>
<td>1,284</td>
<td>1,308</td>
<td>2,148</td>
</tr>
<tr>
<td># occupied Slices</td>
<td>531</td>
<td>663</td>
<td>642</td>
<td>1,004</td>
</tr>
<tr>
<td># BlockRAM/# DSP48E</td>
<td>34/6</td>
<td>34/6</td>
<td>98/6</td>
<td>98/12</td>
</tr>
</tbody>
</table>

Source: EU-Funded FP7 REFLECT project

Strategy 8: 6.8× faster than pure software solution

See: Cardoso et al., Specifying Compiler Strategies for FPGA-based Systems, FCCM 2012
Simple code restructuring example

An FIR
// 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;
}
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;
}

// 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
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;
}

// 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;
}
Code restructuring: FIR example

1
// 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;
}

II=2
1 sample per 2 clock cycles

// 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;
    y[j] = output;
}

II=1
2 samples per clock cycle

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;
    y[j] = output;
    x_3=x[j+1];
    output=c0*x_3;
    output+=c1*x_2;
    output+=c2*x_1;
    output+=c3*x_0;
    y[j+1] = output;
}

1 sample per clock cycle

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

• Manual
  • Programmers need to know the impact of code styles and structures on the generated architecture – with similarities 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
Some approaches for code restructuring/opt.

- Flag selection
- Phase ordering
- Polyhedral models
- Graph-based transformations

- LegUp [Canis et al., ACM TECS’13]: flag selection and phase ordering (via LLVM + opt) [Huang et al., ACM TRETS’15]
- The Merlin Compiler and source to source optimizations by Cong et al., FSP’16
- Polyhedral transformations by Zuo et al., FPGA’13
- Polyhedral in nested loop pipelining by Morvan et al., IEEE TCAD’13
- Graph-based code restructuring by Ferreira and Cardoso, FSP’18, ARC’19
Flag selection

• Generation controlled by enabling/disabling compiler flags – sequence of optimizations are the ones built-in and pre-fixed for each flag
• Suitable to most common approaches, but without taking full-advantage of customization/specialization

Helping but without solving the code restructuring problem!
Phase ordering

- Providing specific sequences of compiler optimizations
- Problem is very complex as besides selecting the phases one needs to provide sequences – usually repeating phases
- Difficult to find the sequence!
- Fully dependent on the portfolio of phases a compiler may include – phases need to justify their inclusion (i.e., if they pay-off)

Limitations for solving the code restructuring problem!
Polyhedral models

• Applied to *Static Control Parts* – require specific loop structures, statically known iteration spaces, limited to affine domains

• Pure polyhedral models transform iteration spaces – more advanced approaches combine the polyhedral model with AST transformations

• Able to provide useful code transformations and justify their inclusion in the portfolio of compiler optimizations

Helping on solving the code restructuring problem!
Graph-based transformations (our ongoing work)

• Traces of computations are represented in Dataflow Graphs (DFGs)
• Code restructuring problem is solved by graph transformations
• Able to achieve high-levels of code restructuring and suitable HLS directives

A proof of concept... scalability still needs to be solved!
Code restructuring: ongoing

Application Code (Software Programming Language) → Analysis, Profiling, Execution → Graphs (e.g., Representing Traces) → Graph-based Optimizations → Code Generation

Input Strategies → Strategies

OpenMP → OpenSPL → OpenCL

LARA DSL → Strategies
Code restructuring: graph-based approach

Application Code (Software Programming Language) → Analysis, Profiling, Execution → DFG (Representing a Trace) → Graph-based Optimizations → Code Generation

Optimize DFG
- Split in subDFGs
- Fold DFGs
- Identify data reuse
- Balance chains of operations
- Data partitioning

+ directives

Configurations

VIVADO
HLx Editions
Example – filter subband

void filter_subband (double z[Nz], double s[Ns], double m[Nm]) {
    double y[Ny];
    int i, j;
    for (i=0; i<Ny; i++) {
        y[i] = 0.0;
        for (j=0; j<(int)Nz/Ny; j++)
            y[i] += z[i+Ny*j];
    }
    for (i=0; i<Ns; i++) {
        s[i] = 0.0;
        for (j=0; j<Ny; j++)
            s[i] += m[Ns*i+j] * y[j];
    }
}

Source: Ferreira and Cardoso, ARC’2019
### Experimental results

- **Vivado HLS 2017.4**
- **Xilinx FPGA Artix-7 (xc7z020clg484-1)**

<table>
<thead>
<tr>
<th>Name</th>
<th>Speedup C</th>
<th>Speedup C-inter</th>
<th>Speedup C-high</th>
<th>Latency (#cycles)</th>
<th>Clock Period (ns)</th>
<th>#LUT</th>
<th>#FF</th>
<th>#DSP</th>
<th>#BRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filter subband</td>
<td>81</td>
<td>5.8</td>
<td>5.8</td>
<td>293 (0.18)</td>
<td>17.1 (0.9)</td>
<td>47537 (7.1)</td>
<td>42589 (3.6)</td>
<td>118 (4.1)</td>
<td>0</td>
</tr>
<tr>
<td>Dotprod</td>
<td>16</td>
<td>5.6</td>
<td>1.0</td>
<td>255 (1)</td>
<td>8.9 (1.0)</td>
<td>294 (1.0)</td>
<td>581 (1.0)</td>
<td>8 (1.0)</td>
<td>0</td>
</tr>
<tr>
<td>Autocorrelation</td>
<td>297</td>
<td>98.6</td>
<td>47.5</td>
<td>16 (0.018)</td>
<td>8.6 (1.1)</td>
<td>8025 (4.0)</td>
<td>7114 (7.9)</td>
<td>160 (16.0)</td>
<td>0</td>
</tr>
<tr>
<td>1D FIR</td>
<td>237</td>
<td>30.0</td>
<td>16.2</td>
<td>120 (0.06)</td>
<td>8.7 (1)</td>
<td>4297 (0.9)</td>
<td>5641 (1.9)</td>
<td>192 (1.6)</td>
<td>0</td>
</tr>
<tr>
<td>2D Convolution</td>
<td>76</td>
<td>5.0</td>
<td>3.0</td>
<td>3886 (0.33)</td>
<td>8.7 (1)</td>
<td>6376 (1.2)</td>
<td>3408 (0.6)</td>
<td>57 (1.5)</td>
<td>0</td>
</tr>
<tr>
<td>SVM</td>
<td>123</td>
<td>3.5</td>
<td>3.5</td>
<td>3208 (0.28)</td>
<td>8.4 (1)</td>
<td>14203 (1.6)</td>
<td>12506 (1.6)</td>
<td>91 (1.6)</td>
<td>76 (1.11)</td>
</tr>
</tbody>
</table>

Source: Ferreira and Cardoso, ARC’2019
Ongoing and future work

• Comparisons to the approaches using the polyhedral model to restructure software code

• Scalability issues
  • How to avoid the need of explicit large graphs when dealing with large traces / loops with many iterations?

• Focus on optimizations regarding conditional paths
  • Use of different execution paths to create specialized accelerators and schemes to manage their execution at runtime
  • Merge of execution paths in order to avoid one specialized accelerator per execution path

Source: Ferreira and Cardoso, ARC’2019
Conclusion

• Source-to-source compilers as front-ends and HLS tools as the new backends for advanced compilation to FPGAs

• Compiling to FPGAs needs more efficient and aggressive code restructuring – a research challenge!

• Our recent efforts focus on an approach to optimize code for HLS based on unfolded graph representations and graph transformations – experimental results highlight the benefits of the approach

• A deeper study about code restructuring approaches needs to be done!
Thank you! Questions?

João MP Cardoso
jmpc@acm.org
Acknowledgments

Afonso Ferreia  
João Bispo  
Pedro Pinto  
Tiago Carvalho  
Luís Reis

HORIZON 2020

ANTAREX

SMILES

CONTEXTWA

PhD scholarships from FCT