A Few Lines of Code, Thousands of Cores:
High-level FPGA Programming using Vector Processor Networks

Wim Vanderbauwhede\textsuperscript{1},
Sai Chalamalasetti\textsuperscript{2}, Sohan Purohit\textsuperscript{2}, Martin Margala\textsuperscript{2}

\textsuperscript{1} School of Computing Science, University of Glasgow, UK
\textsuperscript{2} Dept. of Electrical and Computer Engineering, University of Massachusetts Lowell, Lowell, MA, USA
Outline

- Introduction
  - The MORA Framework
  - Scope of Current Work
- MORA Overview
  - The MORA RC
  - MORA-C++ Features
  - Tool Chain & Compilation
- Specialization and Parallelism
  - RC Specialization
  - RC Vectorization
  - Parallel Program Instances
- Results
The MORA Framework

- Targeted at high performance streaming algorithms.
- Implements a Communicating Sequential Processes (CSP) paradigm.
- Allows memory-based constructs and algorithms.
- Every process is implemented by a compile-time configurable Processor-in-Memory, the MORA Reconfigurable Cell (RC).
- A program is compiled into network of RCs,
- Each RC is tailored to the specific code implementing the process.
This Work:
Extending the framework

Support for:
- Vectorized RCs
- Configurable data path widths
- Parallel instances of the program

Experimental results:
- MORA-C++ on RC-100 (Virtex-4 LX-200)
- Reference implementation in C on Intel Xeon
A C++ API for high-level programming of FPGAs using the MORA framework.

Used to describe the connections between RCs or groups of RCs (modules).

Implements a netlist language using C++ function templates

Main reason for C++ templates is type safety

Any language with similar semantics is suitable the implement the API: C, Java, FORTRAN, OpenCL, ...
RCs are modeled as functions:
\[
\text{rc} :: T_2 \rightarrow T_4 \rightarrow \ldots \rightarrow (T_1, T_2, \ldots)
\]
- input ports ⇒ function arguments
- output ports ⇒ tuple-type return value:
  
  \[
  \text{tuple}<T1,T2,\ldots> \ \text{rc} \ (T3 \ \text{in1}, T4 \ \text{in2}, \ldots) \ \{ \\
  \quad \text{// ...} \\
  \quad \text{tuple}<T1, T2, \ldots> \ \text{res} \ (\text{out1, out2}, \ldots); \\
  \quad \text{return} \ \text{res}; \\
  \}
  \]

Nets are modeled as tuple-type variables:
\[
\text{n}_{34} :: (T_1, T_2)
\]
  
  \[
  \text{tuple}<T1,T2> \ \text{n34} = \text{rc1}(n1,n2,\ldots); \\
  \text{tuple}<T3> \ \text{n5} = \text{rc2}(\text{n34}.\text{get}<1>); \\
  \text{tuple}<T4> \ \text{n6} = \text{rc3}(\text{n34}.\text{get}<2>);
  \]

RC functionality written in plain C/C++

MORA RC is a Harvard-architecture Processor-in-Memory so in principle no restrictions on language features
MORA-C++ Toolchain

MORA-C++ source code → g++ → functional simulation model

MORA-C++ compiler
- Parse into Abstract Syntax Tree
- Generate SSA form
- Allocate memory
- Infer template modules
- Infer split/merge trees
- Build abstract netlist
- Emit MORA assembly

MORA assembler
- Expand generation code
- Flatten hierarchy
- Extract instructions
- Generate netlist
- Create abstract machine model

FPGA back-end
- Generate VHDL files
- Generate toolchain scripts
- Run Xilinx ISE

Simulation back-end
Visualisation back-end

FPGA bitfile
MORA-C++ Compilation

- Compiler written in Haskell
- Compiles MORA-C++ into the MORA assembly language:
  - parsing the Mora-C++ source file into an Abstract Syntax Tree
  - type inference and type checking
  - allocating memory for the arguments of the RC
  - building the netlist (connectivity table)
  - allocating memory across all RCs
  - emitting MORA assembly

- Assembler written in Perl
- Creates a Verilog netlist based on the MORA assembly program
  - parses MORA Assembly language into a set of connected RC objects
  - each object performs an analysis on the instruction code
  - each object emits the Verilog code for the corresponding RC
  - generates a build script for the FPGA toolchain
RC Specialization

- General Template
Specialized Instance
RC Vectorization

- Single Program Instance
Multiple Parallel Instances
SPMD Through Vectorization
Lanes: \([\text{FPGA I/O width}] / [\text{Program I/O width}]\)

DMA Channels

Depends on Program size:

- \#BRAMS used
- \#LUTs used

Assembler can compute the optimal number of lanes and channels from the program
The DCT source code:

typedef Matrix<UChar,8,8> Mat;
const UChar ca[8][8] = { ... };
const Mat c((const UChar**)ca); const Mat ct = c.trans();

Pair<Mat,Nil> dct (Mat a) {
Mat m = c*a*ct; // operator overloading
  Pair<Mat,Nil> res(m,_);
  return res;
}
DCT RC Network

For “Small” DCT
The “small” implementation results in 704 cores, the “fast” in 1056 cores.

This example illustrates how a few lines of code results in a thousand-core vector processor network.
Results

Impact of Data Path Width

Slice Count versus Data Path Width

<table>
<thead>
<tr>
<th># Slices</th>
<th>DWT</th>
<th>DCT (small)</th>
<th>DCT (fast)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8-bit</td>
<td>16-bit</td>
<td>32-bit</td>
</tr>
</tbody>
</table>

[Bar chart showing slice counts for DWT, DCT (small), and DCT (fast) across data path widths of 8-bit, 16-bit, and 32-bit.]
Impact of Vectorization

Comparison of Resource Utilization for 8-bit RCs with/without Vectorization

- **#Slices (K)**
- **#BRAMs**

<table>
<thead>
<tr>
<th></th>
<th>Slice count (K)</th>
<th>BRAM count</th>
</tr>
</thead>
<tbody>
<tr>
<td>DWT, no vec</td>
<td>6</td>
<td>40</td>
</tr>
<tr>
<td>DWT, vec</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>DCT (small), no vec</td>
<td>15</td>
<td>60</td>
</tr>
<tr>
<td>DCT (small), vec</td>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>DCT (fast), no vec</td>
<td>200</td>
<td>180</td>
</tr>
<tr>
<td>DCT (fast), vec</td>
<td>80</td>
<td>60</td>
</tr>
</tbody>
</table>
Throughput vs #Lanes for 8-bit RCs

- DCT (small)
- DCT (Fast)
- DWT
## Results
### DCT Implementation Comparisons

<table>
<thead>
<tr>
<th>Implementation</th>
<th>VHDL</th>
<th>MORA</th>
<th>IJG C code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform</td>
<td>Virtex-5</td>
<td>Virtex-4</td>
<td>Intel Xeon</td>
</tr>
<tr>
<td>Clock speed (MHz)</td>
<td>—</td>
<td>100</td>
<td>2000</td>
</tr>
<tr>
<td>Throughput (MB/s)</td>
<td>1648</td>
<td>4720</td>
<td>224</td>
</tr>
<tr>
<td>Speed-up (HW/SW)</td>
<td>—</td>
<td>21</td>
<td>1</td>
</tr>
<tr>
<td>Throughput@100 MHz</td>
<td>1648</td>
<td>4720</td>
<td>11.2</td>
</tr>
</tbody>
</table>

- MORA-C++ throughput is 4.7 GB/s (maximum system I/O bandwidth 6.4 GB/s)
Conclusions

- Reported the first measured results on an implementation of our high-level FPGA data flow programming framework, MORA-C++.  
- Demonstrated support for variable data path widths  
- Excellent performance for the DCT benchmark: 4.7 GB/s, Virtex-4 LX-200 @100 MHz  
  - Instantiating a network of over a thousand cores,  
  - Using vectorized lanes and DMA channels.  
- Promising approach towards high-level programming of FPGAs for HPC applications.
Future work

- Tool chain
  - OpenCL compiler
    - Adding support for automatic parallelization and vectorization

- Functionality
  - OpenCL framework: scheduling, argument binding
  - External memory access
  - Floating-point computations
Thank you!