Loop Program Mapping and Compact Code Generation for Programmable Hardware Accelerators

Srinivas Boppu, Frank Hannig, and Jürgen Teich

Hardware/Software Co-Design
University of Erlangen-Nuremberg
Germany
## Motivation

### Today's Design Challenges

<table>
<thead>
<tr>
<th>Trends</th>
<th>Requirements</th>
<th>Design Implications</th>
</tr>
</thead>
<tbody>
<tr>
<td>More functions</td>
<td>Higher flexibility</td>
<td>More functions in software</td>
</tr>
<tr>
<td>– multiple standards/modes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Higher complexity</td>
<td>Higher throughput</td>
<td>Parallel processing</td>
</tr>
<tr>
<td>– algorithmic complexity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>– image resolutions, etc.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Constant battery</td>
<td>Higher energy efficiency</td>
<td>More operations per mW</td>
</tr>
<tr>
<td>– battery capacity</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Source: Synopsys Inc
## Design Implications

<table>
<thead>
<tr>
<th>Design Implications</th>
<th>Conclusions</th>
</tr>
</thead>
<tbody>
<tr>
<td>More functions in software</td>
<td>Use standard processors</td>
</tr>
<tr>
<td></td>
<td>- <em>general purpose, DSPs, configurable IP</em></td>
</tr>
<tr>
<td>Parallel processing</td>
<td>Design dedicated hardware</td>
</tr>
<tr>
<td></td>
<td>- <em>highly application-specific</em></td>
</tr>
<tr>
<td></td>
<td>- <em>highly optimized</em></td>
</tr>
<tr>
<td>More operations per mW</td>
<td>Design dedicated hardware</td>
</tr>
<tr>
<td></td>
<td>- <em>highly application-specific</em></td>
</tr>
<tr>
<td></td>
<td>- <em>highly optimized</em></td>
</tr>
</tbody>
</table>

**Solution:** *Custom Programmable Processors*
Overview

• Tightly-Coupled Processor Arrays
• Mapping Flow
• Compact Code Generation
• Results
• Conclusions
• Q&A
Tightly-Coupled Processor Arrays

• Massively parallel processor array
  – Highly customizable at synthesis-time
  – Many parameters and configuration options, e.g.,
    ▪ No. of PEs in the array
    ▪ Within PEs: no. and type of functional units, registers, etc.
  – Reconfigurable interconnect

• Used as *accelerator* in MPSoC for compute-intensive loop programs from domains such as
  – Digital signal processing
  – Image processing, object recognition
  – Linear algebra
  – . . . and other streaming applications

Srinivas Boppu | ASAP13 | June 5-7, 2013 | Washington, DC
Tightly-Coupled Processor Arrays

• Processing elements (PEs)
  - VLIW architecture
  - Weakly-programmable
    ▪ Small instruction set and memory
    ▪ Small register file
    ▪ No direct main memory access

• Zero-overhead static control flow
  - Multiway branch unit in each PE
  - Combined with globally generated control predicates
  - Entire static control flow of the loop program is captured by controller

\[
\text{BRU IF } (\text{IC}_0, \text{IC}_1, \ldots, \text{IC}_{N-1}) \quad \text{add}_0, \text{add}_1, \ldots, \text{add}_{2^N-1}
\]
**Mapping Flow**

**Challenges:**
- Preserving a given mapping and schedule of data flow operations
- Code size should be minimum to meet the tight instruction memory constraints
Mapping Flow: Front End

Example: PAULA program

\[
\begin{align*}
&\text{par}(i=0; \; i<6; \; i++) \\
&\text{par}(j=0; \; j<7; \; j++) \\
&\{ \\
&S_1: a[i,j] = i_0[i,j] + i_1[i,j] \quad \text{if}(j==0) \\
&S_2: a[i,j] = i_2[i,j] + i_3[i,j] \quad \text{if}(1 \leq j \leq 6) \\
&S_3: b[i,j] = a[i,j] \times 7 \\
&S_4: c[i,j] = a[i,j] + b[i,j] \\
&\}
\end{align*}
\]

RDG: Reduced Dependence Graph

Iteration interval (II): time in clock cycles between the evaluation of two successive iterations

All the iterations of the loop are mapped to a single PE(0,0)
Mapping Flow: Front End

Super iteration interval (SII): number of iteration intervals until the same register binding for all variables repeats
Code Generation

\[ j = 0 \]

ADD0  ADD  RD0  ID0  ID1
or
ADD0  ADD  RD2  ID0  ID1

MUL0  MULI  RD1  RD0  7
or
MUL0  MULI  RD1  RD2  7

ADD1  ADD  OD0  RD0  RD1
or
ADD1  ADD  OD0  RD2  RD1

1 \leq j \leq 6

ADD0  ADD  RD0  ID2  ID3
or
ADD0  ADD  RD2  ID2  ID3

ADD1  ADD  OD0  RD0  RD1
or
ADD1  ADD  OD0  RD2  RD1
This method would require \( LL + (\text{iteration space size}) \times II - 1 \) instructions making the generated code dependent on the iteration space size.
Compact Code Generation

• Few Observations...
  − Iteration space can be divided into so called *program blocks* (PBs) due to condition spaces
  − Traversing the iteration space can be represented by *control flow graph*, $G_{CF}$
  − All the iterations that belong to a program block will have the same codes

---

Srinivas Boppu | ASAP13 | June 5-7, 2013 | Washington, DC
Compact Code Generation

• In the first step of code generation approach:
  – For each program block compact code can be generated by contiguously arranging instructions from different iterations --- so called *Overlapped Codes* of PBs.
  
  – In order to have a redundancy in the PB overlapped codes, a certain number of iterations has to be executed in a PB

  – Controller issues appropriate signals so that execution jumps to the right instruction
In the second step of code generation approach:

- **Overlapped Codes** of PBs are arranged contiguously according to the schedule to generate the final compact code
- Control Flow Graph may require some partial unrolling of loops
  - $G_{CF}$ is transformed into a transformed $G_{TCF}$

Our approach needs only 10 instructions compared to fully unrolled code of 44 instructions and it is iteration space size independent.
Results

- Compact Code generation techniques are integrated into an existing frame work called PARO

<table>
<thead>
<tr>
<th>Application</th>
<th>No. of used PEs</th>
<th>Global Latency (#cycles)</th>
<th>code size(^b) (#instr./PE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- 32 taps, 32 samples</td>
<td>4 × 4</td>
<td>262</td>
<td>8</td>
</tr>
<tr>
<td>- 64 taps, 64 samples</td>
<td>4 × 4</td>
<td>1030</td>
<td>8</td>
</tr>
<tr>
<td>- 64 taps, 1000 samples</td>
<td>4 × 4</td>
<td>16006</td>
<td>8</td>
</tr>
<tr>
<td>MM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- 16 × 16 matrix size</td>
<td>2 × 2</td>
<td>102</td>
<td>16</td>
</tr>
<tr>
<td>- 24 × 24 matrix size</td>
<td>3 × 3</td>
<td>130</td>
<td>16</td>
</tr>
<tr>
<td>- 32 × 32 matrix size</td>
<td>4 × 4</td>
<td>158</td>
<td>16</td>
</tr>
<tr>
<td>Gaussian Filtering</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- 100 × 100 image, 3 × 3 mask</td>
<td>3 × 3</td>
<td>10008</td>
<td>14</td>
</tr>
<tr>
<td>- 1000 × 1000 image, 3 × 3 mask</td>
<td>3 × 3</td>
<td>1000008</td>
<td>14</td>
</tr>
</tbody>
</table>

\(^a\) Hardware cost: each PE consists of 3 adders, 3 multipliers, 1 logical unit, 1 shifter, 16 RD, 4 FD, 8 ID, 8 OD registers, and 32 instructions.

\(^b\) this is code size for a single PE but all PEs execute the same program code
Results

• The proposed approach is compared to the results produced by the Trimaran compilation and simulation infrastructure[2]

<table>
<thead>
<tr>
<th>Application</th>
<th>Problem Size</th>
<th>Average Iteration Interval ($I_{avg}$)</th>
<th>% reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Trimaran</td>
<td>Our approach</td>
</tr>
<tr>
<td>FIR</td>
<td>32 filter taps, 1 000 input samples</td>
<td>2.3</td>
<td>1</td>
</tr>
<tr>
<td>MM</td>
<td>multiplication of two $16 \times 16$ matrices</td>
<td>2.5</td>
<td>1</td>
</tr>
<tr>
<td>RE</td>
<td>$M_i = 6$, $M_j = 7$</td>
<td>10.0</td>
<td>1</td>
</tr>
<tr>
<td>SAD</td>
<td>$16 \times 16$ block size</td>
<td>8.4</td>
<td>1</td>
</tr>
</tbody>
</table>

FIR: finite impulse response
MM: matrix multiplication
RE: Running Example
SAD: Sum of absolute differences

*Average Iteration Interval* is the average time between the start of two successive loop iterations. It is calculated by dividing the total execution time of a loop nest by the total number of iterations

Results

<table>
<thead>
<tr>
<th>Application</th>
<th>Problem Size</th>
<th>Number of VLIW Instructions</th>
<th>% reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Trimaran</td>
<td>Our approach</td>
</tr>
<tr>
<td>FIR</td>
<td>32 filter taps, 1,000 input samples</td>
<td>13</td>
<td>8</td>
</tr>
<tr>
<td>MM</td>
<td>multiplication of two $16 \times 16$ matrices</td>
<td>17</td>
<td>16</td>
</tr>
<tr>
<td>RE</td>
<td>$M_i = 6$, $M_j = 7$</td>
<td>18</td>
<td>10</td>
</tr>
<tr>
<td>SAD</td>
<td>$16 \times 16$ block size</td>
<td>14</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Overhead (%)</th>
<th>Trimaran</th>
<th>Our approach</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>14.1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>26.4</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>11.1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>4.6</td>
<td>0</td>
</tr>
</tbody>
</table>

*Overhead* indicates the amount of time that is spent in executing other than the inner most loop compared to the total execution time
Our approach for code generation

- Preserves the given scheduling of data flow operations

- Minimizes the code size to meet the tight instruction memory constraints; thus, also reducing the area and power

- Code size is also independent of the iteration space size
Contact

Srinivas Boppu
Hardware/Software Co-Design, Department of Computer Science
University of Erlangen-Nuremberg
Cauerstr. 11, 91058 Erlangen, Germany
Email: srinivas.boppu@fau.de    Phone: + 49 9131 85-67315