REGISTER ALLOCATION FOR HIGH-LEVEL SYNTHESIS OF HARDWARE ACCELERATORS TARGETING FPGAS

Gerald Hempel, Christian Hochberger, Jan Hoyer, Thilo Pionteck

Darmstadt, 12 July 2013
Outline

- Motivation
- Design Flow
- Register Allocation
- Results & Discussion
- Summary
Motivation

- FPGAs suitable platform for application-specific designs
- Predominant concept → standard processor IP-core in combination with application-specific accelerators
- Goal: Automatic mapping of high level application code into HW
  - Synthesis result is a combination of HLS and vendor synthesis tools (XST, Synopsis, Altera Synthesizer)
Motivation

- FPGAs suitable platform for application-specific designs
- Predominant concept → standard processor IP-core in combination with application-specific accelerators
- Goal: Automatic mapping of high level application code into HW
  - Synthesis result is a combination of HLS and vendor synthesis tools (XST, Synopsis, Altera Synthesizer)

**How much effort is required in HLS optimization?**

- Evaluation of several register allocation strategies for FPGA based accelerators on Spartan 6 and Artix 7
- Underlying synthesis tool: Xilinx XST P.49d in ISE 14.4
SpartanMC Soft-Core and Kernel Interface

- SpartanMC processor soft-core for software execution
- Program and data stored in local BRAM
- HW accelerators treated as peripherals
- Peripheral stub used as wrapper for accelerator
- Peripheral stub provides access to memory and peripheral bus
  - Parameters transferred via peripheral bus during kernel startup
  - Direct BRAM accesses (triggered by pointers and arrays) during kernel execution
Typical Workflow

Application Sources

Compiler

Intermediate Representation

HDL

FPGA Vendor Synthesis Tools

Binary Objects

Accelerator Logic

Target FPGA

Analysis Data

Processor
Typical Workflow

- Application Sources
- Analysis Data
- GCC 4.4.5
- Intermediate Representation
- GIMPLE
- HDL
- Binary Objects
- FPGA Vendor Synthesis Tools
- Xilinx XST P.49d
- Accelerator Logic
- Processor SpartanMC Soft-Core
- Target FPGA

Darmstadt, 12 July 2013
Kernel Extraction using GCC

- GIMPLE passes for analysis and synthesis
Kernel Extraction using GCC

- **GIMPLE passes for analysis and synthesis**

  ![Diagram](image)

  - **Analysis:**
    - Finds most worthy loop
    - Omits functions calls (inlined functions only)
Kernel Extraction using GCC

- GIMPLE passes for analysis and synthesis

- Analysis:
  - Finds most worthy loop
  - Omits functions calls
    (inlined functions only)

- Synthesis:
  - Uses list scheduling
  - Performs high-level register allocation
int foo (int a, int b) {
    int i;
    for (i=0; i<100; i++) {
        a = a * a;
        b = a - b * i;
        if (b < 0)
            b = b * -1;
    }
    return b;
}
```c
int foo (int a, int b) {
    int i;
    for (i=0; i<100; i++) {
        a = a * a;
        b = a - b * i;
        if (b < 0)
            b = b * -1;
    }
    return b;
}
```
Register Allocation Strategies

• **le_simple**: Assigns a register for each GIMPLE-variable

```
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Range</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_8</td>
<td>1.0-3.0</td>
<td>(global)</td>
</tr>
<tr>
<td>D1_9</td>
<td>1.0-1.1</td>
<td>(local)</td>
</tr>
<tr>
<td>b_10</td>
<td>1.1-1.2</td>
<td>(local)</td>
</tr>
<tr>
<td>b_2</td>
<td>1.2-3.0</td>
<td>(global)</td>
</tr>
<tr>
<td>b_11</td>
<td>2.0-2.1</td>
<td>(local)</td>
</tr>
<tr>
<td>i_13</td>
<td>3.0-3.1</td>
<td>(local)</td>
</tr>
<tr>
<td>a_19</td>
<td>3.1-1.0</td>
<td>(global, loop)</td>
</tr>
<tr>
<td>b_20</td>
<td>3.1-1.0</td>
<td>(global, loop)</td>
</tr>
<tr>
<td>i_21</td>
<td>3.1-3.0</td>
<td>(global, loop)</td>
</tr>
</tbody>
</table>
```

![Diagram showing register allocation strategies](slide8.png)
Register Allocation Strategies

- **le_full**: Minimize the number of registers

```plaintext
a_8: 1.0-3.0 (global)
D1_9: 1.0-1.1 (local)
b_10: 1.1-1.2 (local)
b_2: 1.2-3.0 (global)
b_11: 2.0-2.1 (local)
i_13: 3.0-3.1 (local)
a_19: 3.1-1.0 (global, loop)
b_20: 3.1-1.0 (global, loop)
i_21: 3.1-3.0 (global, loop)
```
Register Allocation Strategies

- **le_uid**: Maps variables with identical GIMPLE-ID to one register

<table>
<thead>
<tr>
<th>Variable</th>
<th>Range</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_8</td>
<td>1.0-3.0</td>
<td>(global)</td>
</tr>
<tr>
<td>D1_9</td>
<td>1.0-1.1</td>
<td>(local)</td>
</tr>
<tr>
<td>b_10</td>
<td>1.1-1.2</td>
<td>(local)</td>
</tr>
<tr>
<td>b_2</td>
<td>1.2-3.0</td>
<td>(global)</td>
</tr>
<tr>
<td>b_11</td>
<td>2.0-2.1</td>
<td>(local)</td>
</tr>
<tr>
<td>i_13</td>
<td>3.0-3.1</td>
<td>(local)</td>
</tr>
<tr>
<td>a_19</td>
<td>3.1-1.0</td>
<td>(global,loop)</td>
</tr>
<tr>
<td>b_20</td>
<td>3.1-1.0</td>
<td>(global,loop)</td>
</tr>
<tr>
<td>i_21</td>
<td>3.1-3.0</td>
<td>(global,loop)</td>
</tr>
</tbody>
</table>
Benchmarks

- Typical algorithms for embedded systems:
  - base64 encoder, bitreverse, FFT, grayscale filter, IIR filter, haar wavelet transformation, matrix multiplication
- Kernels were generated automatically for compute intensive parts
- Parameter sweep:
  - *Area* or *speed* optimization
  - *Spartan 6* or *Artix 7* device
  - Allocation strategy *(le_full, le_simple, le_2, le_3, le_4, le_5, le_uid)*
  - 8 benchmarks
- 224 test cases
Artix 7 Resource Consumption

Used LUTs (optimized for area)
Artix 7 Resource Consumption

- Allocation strategies show little effect on resource consumption
- Best results for le_simple and le_uid

Used LUTs (optimized for area)
Artix 7 Frequency Results

Achievable clock frequency
(optimized for area)
Artix 7 Frequency Results

- Results for le_2, le_3, le_4, le_5 hard to predict (s. haar wavelet, bit reverse and base 64 encoder)
- FFT, grayscale filter, IIR filter and matrix multiplication → negative effect for all allocation strategies except le_uid and le_simple

Achievable clock frequency (optimized for area)

Darmstadt, 12 July 2013
## What is the difference?

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>#GIMPLE variables</th>
<th>#basic blocks</th>
<th>#branches</th>
<th>#ALU operations</th>
<th>#MUL operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>base64 encode</td>
<td>50</td>
<td>3</td>
<td>1</td>
<td>21</td>
<td>0</td>
</tr>
<tr>
<td>bitreverse</td>
<td>62</td>
<td>22</td>
<td>11</td>
<td>61</td>
<td>0</td>
</tr>
<tr>
<td>bitreverse (spec.)</td>
<td>62</td>
<td>6</td>
<td>3</td>
<td>61</td>
<td>0</td>
</tr>
<tr>
<td>haar wavelet</td>
<td>14</td>
<td>3</td>
<td>1</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>FFT</td>
<td>51</td>
<td>6</td>
<td>2</td>
<td>22</td>
<td>4</td>
</tr>
<tr>
<td>grayscale filter</td>
<td>29</td>
<td>3</td>
<td>1</td>
<td>17</td>
<td>2</td>
</tr>
<tr>
<td>IIR filter</td>
<td>79</td>
<td>6</td>
<td>2</td>
<td>29</td>
<td>11</td>
</tr>
<tr>
<td>matrix mult</td>
<td>59</td>
<td>3</td>
<td>1</td>
<td>25</td>
<td>8</td>
</tr>
</tbody>
</table>
What is the difference?

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>#GIMPLE variables</th>
<th>#basic blocks</th>
<th>#branches</th>
<th>#ALU operations</th>
<th>#MUL operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>base64 encode</td>
<td>50</td>
<td>3</td>
<td>1</td>
<td>21</td>
<td>0</td>
</tr>
<tr>
<td>bitreverse</td>
<td>62</td>
<td>22</td>
<td>11</td>
<td>61</td>
<td>0</td>
</tr>
<tr>
<td>bitreverse (spec.)</td>
<td>62</td>
<td>6</td>
<td>3</td>
<td>61</td>
<td>0</td>
</tr>
<tr>
<td>haar wavelet</td>
<td>14</td>
<td>3</td>
<td>1</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>FFT</td>
<td>51</td>
<td>6</td>
<td>2</td>
<td>22</td>
<td>4</td>
</tr>
<tr>
<td>grayscale filter</td>
<td>29</td>
<td>3</td>
<td>1</td>
<td>17</td>
<td>2</td>
</tr>
<tr>
<td>IIR filter</td>
<td>79</td>
<td>6</td>
<td>2</td>
<td>29</td>
<td>11</td>
</tr>
<tr>
<td>matrix mult</td>
<td>59</td>
<td>3</td>
<td>1</td>
<td>25</td>
<td>8</td>
</tr>
</tbody>
</table>

- Multiply-accumulate operation are mapped to sequential DSP primitives
- Multiplexer tree caused by variable swaps leads to performance degradations
Artix 7 normalized Frequency

- Achievable clock frequency optimized for area normalized to one
Artix 7 normalized Frequency

- Achievable clock frequency optimized for area normalized to one
- Gap of about 30% between le_simple, le_uid and more complex algorithms
Conclusion

For HLS targeting FPGAs:

- Complex registers allocation strategies resulting in a reuse of a single register for many variables are not advisable
  - Little effect on area consumption
  - May lead to performance degradations (up to 30%)

- Naive allocation strategy gives most freedom to synthesis tool and mostly better results
Artix 7 normalized LUTs

- A simple allocation strategy inclines a small resource footprint.


Artix 7 Frequency and LUTs (speed)

- Achievable clock frequency optimized for speed
- Used LUTs optimized for speed

Darmstadt, 12 July 2013
Register Allocation
Spartan 6 Frequency and LUTs (area)

- Achievable clock frequency optimized for area
- Used LUTs optimized for area
Spartan 6 Frequency and LUTs (speed)

- Achievable clock frequency optimized for speed
- Used LUTs optimized for speed