From Low-architectural Expertise Up to High-throughput Non-binary LDPC Decoders: Optimization Guidelines using High-level Synthesis

João Andrade¹, Nithin George², Kimon Karras³, David Novo¹, Vitor Silva¹, Paolo Ienne², Gabriel Falcão¹

¹ University of Coimbra, PT; ² EPFL, CH; ³ Xilinx Research Labs, IE
Outline

The Challenge

The Problem

Non-binary LDPC decoding

Decoding architecture

HLS decoder design

Experimental Results

Conclusion
The Challenge
Design an efficient LDPC decoder fast

- RTL requires too specialized knowledge
- Our background is GPU and not hardware
- Error-prone design space exploration (DSE)
- Extensive code refactoring for DSE
- High-level synthesis (HLS) has been around for years
- Fast time to market
Design an efficient LDPC decoder fast

- Adjust DS decisions much faster wo/ extensive refactoring
  - Bitwidth for a particular SNR operation point
  - Decoding schedule
  - Decoding algorithm

- C/C++ code base can be used with Vivado HLS
  - C/C++ supported
  - Cycle-accurate simulation after C-synthesis
  - Code annotations (#pragma) or Tcl commands

Why?

- Power budgets of GPUs way above requirements

- Real-time operation is required
  - High decoding throughputs
  - Low latencies
The Problem
FEC on communication systems

- Belief propagation problem $\rightarrow$ LDPC codes decoding
- Non-binary LDPC can tackle
  - Quantum-key distribution
  - Erasure channel (burst)
  - AWGN channel
- But have a very high (non-linear) numerical complexity
- Irregular data patterns and intensive access profile
Soft-Decoding Algorithms

Belief-Propagation I

- LDPC decoding is a particular case of belief-propagation

- Messages circulate through a bipartite graph structure with computation applied at the node- and edge-level
Soft-Decoding Algorithms
Belief-Propagation II

- The bipartite model can be employed to generalize other algorithms w/ the following constraints
  - node level functions → must produce/consume data coherently
  - edge level functions → produce/consume without restrictions

- By defining these kernels different algorithms can be defined
  - CN and VN → Hadamard products
  - Edges permute/depermute
  - Edges apply the Fast Walsh-Hadamard Transform
HLS decoder
Mapping the LDPC Tanner graph

- We attempt an isomorphic transformation of the Tanner graph

- Therein, each node/edge-level kernel become their own C-function and nodes/edges an iteration within a loop structure
HLS decoder

Where to begin?

- There are several dimensions to non-binary LDPC decoding
  - (code related)
  - \( N \) VNs and \( M \) CNs to process
  - Each VN connects to \( d_v \) CNs
  - Each CN connects to \( d_c \) VNs
  - (Galois Field related)
  - \( 2^m \) probabilities to compute per probability mass-function (pmf)
HLS decoder
How to express computation?

• Suppose a GPU SIMT-architecture mindset
  
  //flat loop unsuitable for Vivado HLS optimizations
  for(int i = 0; i < edges*q*d_v; i++){
    int e = i/(d_v*q); //get VN id
    int g = i%q; //get GF(q) element
    int t = (i/q)%d_v; //get d_v element

    computation();
  }

• What is it any different than this?
  
  //nested loop suitable for Vivado HLS optimizations
  for(int e = 0; e < edges; e++)
    for(int g = 0; g < q; g++)
      for(int t = 0; t < d_v; t++)

        computation();

• Optimizations are hardly picked up in the former
HLS decoder
Loop structures

- Loop trip counts
- E: edges or $N \times d_v = M \times d_c$
- GF: $2^m$
- LOGGF: $m$
- Dv/Dc: $d_v / d_c$
- Local BRAM copies are maintained
- Data streams from DRAM
HLS decoder
Loop structures

//nested loop structure of vn_proc
E: for(int e = 0; e < edges; e++)
   GF: for(int g = 0; g < q; g++)
      Dv: for(int t = 0; t < d_v; t++)
         //computation follows
HLS decoder

Loop structures

```c
E: for(int e = 0; e < limit; e++){
    GF_read: for(int g = 0; g < GF; g++)
    //load data into temporary buffer
    GF_write: for(int g = 0; g < GF; g++)
    //permute and store back to memory
}
```
HLS decoder
Loop structures

```c
E: for(int e = 0; e < edges; e++){
    G_read: for(int g = 0; g < q; g++){
        //load data into temporary array
    }
    LogGF: for(int c=0; c<m; c++)
    GF: for(int g = 0; g < q; g++)
        //perform Radix-2 computation
    G_write: for(int g = 0; g < q; g++){
        //store data back to memory
    }
}
```

---

**CN->VN**
**VN->CN**
**E:** edges
**GF:** $2^m$
vn_proc permute
**E:** edges
**E:** edges
**LogGF:** m
fwht
cn_proc depermute
**E:**
**G:** ...
partitioned in Solutions IV-VII
**E:** edges
**LogGF:** m
GF_read: $2^m$
**GF:** $2^m$
GF_write: $2^m$
2 RW ports available
HLS decoder solutions

Partitioning

- Scheduling analysis after C-synthesis shows lack of mem. ports
- BRAMs are instantiated with dual-port control
- Further action is required

```python
set_directive_resource -core RAM_T2P_BRAM
set_directive_array_partition -type cyclic -factor 4 -dim 1
```
HLS decoder

Optimization solution I

- Solution I, base version w/o optimizations
Solution II Full unroll of inner loops LOGGF and GF

set_directive_unroll "*/LOGGF"
set_directive_unroll "*/GF"
HLS decoder
Optimization solution III

- Solution III, II+pipeline of outer loops E to II=1
  set_directive_pipeline "*/E" -II=1
Solution IV, I+cyclic partitioning of all BRAM arrays by a factor of $2^m$

set_directive_array_partition -type cyclic -factor $2^m$ -dim 1 "decoder"
Solution V, IV+full unroll of inner loops LOGGF and GF
Solution VI, III+IV (unroll, pipeline, partition)
HLS decoder
Optimization solution VII

- Solution VII, IV+pipeline of inner loops LOGGF GF to II=1

```plaintext
set_directive_pipeline "*/LOGGF" -II=1
set_directive_pipeline "*/GF" -II=1
```
HLS decoder

- Define fixed-point computation suitable for a target SNR/BER

```c++
#include<ap_cint.h>
//data is stored in llr type variables
//computation is performed in llr_ type variables
//use floating-point
typedef float llr;
typedef float llr_; //use Q8.7 fixed-point
typedef ap_fixed< 8, 1, AP_RND_INF, SC_SAT > llr;
typedef ap_fixed< 16, 3, AP_RND_INF, SC_SAT > llr_;
```
Experimental results
Clock frequency and latency

• Best clock frequency of operation obtained for Solution VI
• Lowest latency always achieved for Solution VI
• Solution III is a good compromise between VI and the remaining Solutions
• Solution VII replication of pipelined loops is a poor design choice
  • Most alike to OpenCL strategy
Experimental results

FPGA utilization

- Under 20% LUT util.
- Multiple decoder instantiation
- What about pin, clock and mem interface?
HLS host platform

- Originally RTL-project and can be Tcl’d automatically
- VC709 board target (693K CLBs)

- Two DRAM banks controlled (MIG)
- Two AXI interconnect can be configured for up to 16 HLS cores
- Data streams from the DRAM bank 0 and to bank 1
- Each HLS core performs computation to its own “BRAM” space

- DMA via PCIe → 3K LUTs
The host HLS arch and the multiple decoders elevate the LUT utilization to $\sim 80\%$

- $\{14, 5, 3\}$ decoders for $GF(2^2)$, $GF(2^3)$ and $GF(2^4)$
### Experimental results

**Comparison with RTL decoders**

<table>
<thead>
<tr>
<th>Decoder</th>
<th>m</th>
<th>K</th>
<th>LUT [%]</th>
<th>FF</th>
<th>BRAM</th>
<th>DSP</th>
<th>Thr. [Mbit/s]</th>
<th>Clk [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>This work</td>
<td></td>
<td></td>
<td>14</td>
<td>7</td>
<td>0.5</td>
<td>0.5</td>
<td>1.17</td>
<td>250</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td>80</td>
<td>35</td>
<td>6</td>
<td>6</td>
<td>14.54</td>
<td>219</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1</td>
<td>21</td>
<td>9</td>
<td>0.9</td>
<td>0.9</td>
<td>0.95</td>
<td>250</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>6</td>
<td>81</td>
<td>34</td>
<td>5</td>
<td>5</td>
<td>4.81</td>
<td>210</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1</td>
<td>30</td>
<td>13</td>
<td>2</td>
<td>2</td>
<td>0.66</td>
<td>216</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>3</td>
<td>73</td>
<td>32</td>
<td>5</td>
<td>5</td>
<td>1.85</td>
<td>201</td>
</tr>
<tr>
<td>Zhang TCS–I’11</td>
<td>4</td>
<td></td>
<td>48 (Slices)</td>
<td>41</td>
<td>–</td>
<td>–</td>
<td>9.3</td>
<td>–</td>
</tr>
<tr>
<td>Emden ISTC’10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>–</td>
<td></td>
<td>33.16</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>–</td>
<td></td>
<td>13.22</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td>–</td>
<td></td>
<td>1.56</td>
<td></td>
</tr>
<tr>
<td>Spagnol SiPS’09</td>
<td>3</td>
<td></td>
<td>13</td>
<td>3</td>
<td>1</td>
<td>–</td>
<td>≤4.7</td>
<td>99</td>
</tr>
<tr>
<td>Boutillon TCS–I’13</td>
<td>6</td>
<td></td>
<td>19</td>
<td>6</td>
<td>1</td>
<td>–</td>
<td>2.95</td>
<td>61</td>
</tr>
<tr>
<td>Andrade ICASSP’14</td>
<td>8</td>
<td></td>
<td>85 (LEs)</td>
<td>62</td>
<td>7</td>
<td></td>
<td>1.1</td>
<td>163</td>
</tr>
<tr>
<td>Scheiber ICECS’13</td>
<td>1</td>
<td></td>
<td>14 (Slices)</td>
<td>21</td>
<td>–</td>
<td></td>
<td>13.4</td>
<td>122</td>
</tr>
</tbody>
</table>

* Differences in technology nodes and FPGA are not considered.
Comparison with previous HLS works

- Maxeler decoders allow for $\sim$1 Gbit/s decoding throughputs
  Pratas, GlobalSIP’13
  Andrade, ASAP’14

- OpenCL (Altera) decoder peaks at hundreds of Kbit/s
  Andrade, ICASSP’14

<table>
<thead>
<tr>
<th></th>
<th>FPGA Util.[%]</th>
<th>$\text{GF}(2^3)$</th>
<th>$\text{GF}(2^3)$ (floating-point)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LUTs</td>
<td>I</td>
<td>IV</td>
</tr>
<tr>
<td></td>
<td>0.64</td>
<td>1.13</td>
<td>5.20</td>
</tr>
<tr>
<td></td>
<td>FF</td>
<td>0.28</td>
<td>0.53</td>
</tr>
<tr>
<td></td>
<td>DSP</td>
<td>0.06</td>
<td>0.06</td>
</tr>
<tr>
<td></td>
<td>BRAM</td>
<td>0.44</td>
<td>0.82</td>
</tr>
</tbody>
</table>

- Vivado HLS decoder reaches dozens of MBit/s
  Scheiber, ICECS’13
Summary

- Code writing-style counts
  - Language is the same, model is not

- Design optimizations come hand-in-hand with the code writing-style
  - Clearly defined bounds are better
  - Optimizations can be double-edge swords

- We can achieve same ballpark figures of RTL
  - Higher utilization

Outlook

- When will platforms be automatically generated?
- When will the C programming model merge?
*(b++) = *(a++) * c;

b[i] = a[i] * c;

Qué?

Ah, sí!
Thank you. Questions are welcome.
What tool to use?

What HLS tool?

• How much are we willing to lose in control?
  • A lot? → OpenCL (C-based)
  • Dataflow? → MaxCompiler (JAVA)
  • Some? → LegUp, Vivado HLS (C/C++, SystemC)
  • None? → Stick to RTL (Verilog, VHDL)

• Vivado HLS allows fine control over
  • Loop scheduling → unroll, pipeline, merge, flatten
  • AXI4 blocks → master/slave memory and stream interfaces
  • Arbitrary bitwidth → fixed-point types supported
  • No clock, no external memory interfaces, and no pin I/O layout