High-Level Synthesis of Dynamic Data Structures: A Case Study Using Vivado HLS

Felix Winterstein | Samuel Bayliss | George Constantinides
f.winterstein12@imperial.ac.uk

Abstract
Recent evaluations report that C-to-FPGA flows can produce results with a quality close to hand-crafted designs [1]. Programs which use dynamic, pointer-based data structures remain difficult to implement well, yet these structures are common in software. We compare a data-flow centric implementation to a recursive tree traversal implementation which makes use of pointer-linked data structures. We present source code transformations that ensure synthesisability and improve the quality of results. The automation of these transformations motivates future research.

Case Study
- Compare two equivalent algorithms for K-means clustering: Lloyd’s Algorithm and the Filtering Algorithm [2]
- Determine required source code refactoring

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Lloyd’s Algorithm</th>
<th>The Filtering Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clustering result</td>
<td>Both algorithms produce the same result</td>
<td>210 - 10^4 node-centre interactions</td>
</tr>
<tr>
<td>Algorithm complexity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Computational properties</td>
<td>Simple control flow</td>
<td>Uses tree structure built from the data set</td>
</tr>
<tr>
<td></td>
<td>Embarrassingly parallel</td>
<td>Recursive tree traversal</td>
</tr>
<tr>
<td></td>
<td>Well-suited for FPGAs</td>
<td>Dynamic memory allocation</td>
</tr>
<tr>
<td>Implementation in Vivado HLS</td>
<td>Seamless C-to-FPGA implementation</td>
<td>Requires substantial source code modifications</td>
</tr>
</tbody>
</table>

Results
- Taken from placed and routed FPGA designs
- Lloyd’s implementation achieves ‘close to hand-written’ performance using synthesis directives only
- > 5 x Improvement of latency due to manual source code modifications for the Filtering Algorithm

4 Code Transformations - Filtering Algorithm
1. On-chip dynamic memory allocation: Custom implementation of the allocator
2. Standard approach to handle recursion: Replaced by a while-loop and a stack
3. Manual partitioning of the tree data structure, privatisation of heap memory for centre sets and memory allocators
4. Loop distribution to enable efficient pipelining

Why Pointers?
- Unbalanced, data-dependent tree shape:
- Pointer-linked data structure
- Dynamic instead of static memory allocation:
  - Significant on-chip memory savings (reuse memory space)

Conclusion
- State-of-the-art HLS tools can achieve ‘close to hand-written’ performance
- Lack of automated optimisations for programs using pointers and dynamic structures: extensive code transformations required
- Future work focuses on a dependence / disjointness analysis for data structures accessed through pointers in order to automate these code transformations

References

Acknowledgements
This work is sponsored by the European Space Agency under the Networking/Partnering Agreement No. 4000106443/12/D/JR and by the EPSRC under grant numbers EP/L000307 and EP/L001206.