Abstract

Cores for non-fewer resources at the expense of more BRAMs used and lower clock Experiments acceptable approximation error.

Operators used in neural network applications are often not available in FPGA vendor’s core library, or may be composite operators requiring several elementary operations.

- We built an open-source, parameterized floating-point core generator, NnCore, for operators used as activation functions, and their derivatives.
- We propose a binary search algorithm to search for minmax-polynomial segments, with step to ensure monotonicity between adjacent segments.
- Input parameters are exponent and mantissa bit-width, and the acceptable approximation error.

Experiments show that, generated cores can require lower latency, fewer resources at the expense of more BRAMs used and lower clock achieved; or they could require higher latency but comparable or less resources and clock achieved, when compared with composed cores written in HLS C++.

Cores for non-standard floating-point widths are supported, hence allowing exploration on different number formats at application level.

Core Designs

- The NnCore generates ReLU, ReLU6, Tanh, Sigmoid, d’Tanh and d’Sigmoid operators.
- Except for ReLU and ReLU6, the segmentor is a naive implementation, where for n polynomial segments we use n comparators, and this n-bit output is feed to an encoder to generate the corresponding coefficient ROM address.
- The Pros of this scheme is its simplicity, and that it does not require any ROM space for the segment boundaries \( b_{\text{up}} \). Also, the latency of the circuit does not increase as number of segments increase in this scheme. As the number of segment from the algorithm is small such scheme is viable.
- It also allows optimizations of the comparators during synthesis since the boundaries are hardwired.

NnCore itself is written in Python, where generated cores are in Verilog, with floating-point adder, multiplier and squarer(odd-power) variant from the FloPoCo project being instantiated.

Polynomial evaluation is done in the simple Horner’s form, e.g.: \( p(y) = a_0 + x(a_1 + x(a_2 + \ldots + x(a_{d-1} + a_dx))) \).

- The all-power and odd-power-only variants only differs in the need of a squarer, and the number of adder/multipliers.

Segment Search Algorithm

1. i ← 0, max_err ← 0, old_max_err ← \( \infty \), mono_steps ← 1, 2. shrink dir ← SHRINK
3. shrink ← False, expand ← False, commit ← False
4. Slist ← Ø, Tlist ← Ø
5. \( S \leftarrow [0; \text{bound}_{\text{up}}] \) if \( f \) is odd or even \( \text{[bound}_{\text{low}}; \text{bound}_{\text{up}}] \) otherwise
6. while not done ∧ not failed 7. if not shrink ∧ not expand 8. poly ← get poly(S)
9. if poly = Ø then \( \text{[shrink} ← \text{True] \)}
10. max_err ← check err_in_range binade(S, poly)
11. if max_err < target 12. mono ← check mono(S, poly, Slist)
13. if not mono 14. \( S \leftarrow \left[ S_{\text{up}} = \frac{S_{\text{up}} - S_{\text{low}}}{\text{mono_steps}} \right] \) 15. mono_steps ← mono_steps + 1 16. continue
17. else
18. if Tlist ≠ Ø then \( \text{[Tlist ← Ø]} \)
19. Tlist ← Tlist US
20. expand ← True 21. else
22. if not max_err < old_max_err then \( \left\{ \text{shrink dir} \right\} \)
23. if shrink dir is SHRINK then (shrink ← True)
24. else (expand ← True)
25. old_max_err ← max_err
26. if expand \( \left[ \text{expand seg}(S) \right] \)
27. if shrink \( \left[ \text{shrink seg}(S) \right] \)
28. if commit 29. if Tlist ≠ Ø 30. \( \text{[s}_{\text{up}} = \text{bound}_{\text{up}} \text{where s = [s}_{\text{low}}; s_{\text{up}}] \} \) ∈ Tlist \( \text{done} ← \text{True] \)
31. Slist ← Slist ∪ Tlist 32. Tlist ← Ø 33. else 34. failed ← True

Experimental Results

- Experiment target were set to an Alpha-Data ADM-PCIE-7V3 board, featuring a Xilinx xc7v690tf691-2 chip, target clock speed = 250MHz.
- Results for TANH at \( a_{\text{up}} = 8 \), \( a_{\text{up}} = 23 \), approx_err = 1 ULP

<table>
<thead>
<tr>
<th>Latency</th>
<th>WNS(Ms)</th>
<th>CLK(MHz)</th>
<th>LUTs</th>
<th>FFs</th>
<th>DSPs</th>
<th>BRAMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>HLS(sinh/cosh)</td>
<td>93</td>
<td>0.247</td>
<td>266</td>
<td>18322</td>
<td>10246</td>
<td>30</td>
</tr>
<tr>
<td>HLS(exp)</td>
<td>42</td>
<td>0.249</td>
<td>267</td>
<td>3365</td>
<td>2494</td>
<td>18</td>
</tr>
<tr>
<td>NnCore(d=3)</td>
<td>34</td>
<td>-1.049</td>
<td>198.1</td>
<td>1742</td>
<td>1498</td>
<td>6</td>
</tr>
<tr>
<td>NnCore(d=5)</td>
<td>61</td>
<td>0.183</td>
<td>262</td>
<td>2747</td>
<td>2843</td>
<td>10</td>
</tr>
</tbody>
</table>

- At single-precision (no denormal numbers), generated Tanh cores can reach 1-ULP (unit of least precision) approximation error.
- At max polynomial degree 3, the core has lower latency (cycles), requires fewer LUTs, flip-flops and DSPs, but at the expense of using some BRAMs and a lower achieved clock rate.
- At max polynomial degree 5, the core still uses less LUTs and DSPs, and no BRAMs, also a comparable achieved clock rate, but requires higher latency due to the Horner form polynomial evaluation.

- Results of generated cores with ODD-POWERS ONLY, at \( a_{\text{up}} = 5 \), \( a_{\text{up}} = 10 \)

<table>
<thead>
<tr>
<th>Deg</th>
<th>Err(ULP)</th>
<th>Lat</th>
<th>WNS(ns)</th>
<th>CLK(MHz)</th>
<th>LUTs</th>
<th>FFs</th>
<th>DSPs</th>
<th>BRAMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tanh</td>
<td>3</td>
<td>21</td>
<td>0.124</td>
<td>258.0</td>
<td>541</td>
<td>579</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Sigmoid</td>
<td>3</td>
<td>21</td>
<td>0.035</td>
<td>252.2</td>
<td>650</td>
<td>589</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>d’Tanh</td>
<td>3</td>
<td>21</td>
<td>0.014</td>
<td>250.9</td>
<td>574</td>
<td>584</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>d’Sigmoid</td>
<td>3</td>
<td>21</td>
<td>0.149</td>
<td>259.7</td>
<td>631</td>
<td>583</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

- At half-precision (no denormal numbers), generated cores requires no BRAMs with degree 3 odd-power polynomials.
- Such operators wouldn’t have been available without our NnCore generator in Xilinx Vivado HLS, since only elementary operators \(+, -, \times, \div\) are available but not the \exp or sinh/cosh required in the above operators.