HLS4ML Progress Update

Parallel Computing System Lab
National Yang Ming Chiao Tung University
Our Interests and Goals

● SW/HW design, optimization, architectures
● Enhancing GNN implementation
● Devise optimizations for GNN HLS design
● Explore tradeoffs of GNN accelerators and propose a possible general optimization method for GNNs on HLS4ML
Outline

- Current Progress on HLS4ML
- Plan for Next Steps
- Questions and Discussion
Current Progress on HLS4ML

- Pass synthesis with example network
- Perform preliminary exploration
Dataflow of HLS4ML

1. Trained Python model (with ml libraries)
2. HLS4ML
3. Cpp code
4. HLS tool (Vivado)
5. Synthesizable RTL
HLS4ML Tool Summary

1. Read the input model and transform to hls4ml object
2. Config the each layer
3. HLS writer will transform the model layer to corresponding C++ hls code
4. Use vivado_hls to synthesize
What We Have Done

1. Understanding how hls4ml transform model to HLS code
2. Run example project
3. Analyze the latency and resource utilization of generated architecture
4. Adjust the reuse factor and compare the result
5. Run gnn_simple project
6. Adjust the graph parameter and successfully synthesis
7. Understanding the template c++ HLS code
Run Example Project

hls4ml-tutorial/part1_getting_started.ipynb
### Compare the Resource between Different Reuse Factors

```python
config = hls4ml.utils.config_from_keras_model(model, granularity='model', default_reuse_factor=1)
```

<table>
<thead>
<tr>
<th>Module</th>
<th>BRAM_18KDSP48E</th>
<th>FF</th>
<th>LUT</th>
<th>URAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>dense_latency_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_1</td>
<td>0</td>
<td>18611124935824</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>dense_latency_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_2</td>
<td>0</td>
<td>979</td>
<td>614518240</td>
<td>0</td>
</tr>
<tr>
<td>dense_latency_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_s</td>
<td>0</td>
<td>920</td>
<td>612918012</td>
<td>0</td>
</tr>
<tr>
<td>dense_latency_ap_fixed_ap_fixed_config11_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0</td>
<td>0</td>
<td>160</td>
<td>961</td>
<td>2825</td>
</tr>
<tr>
<td>relu_ap_fixed_ap_fixed_16_6_5_3_0_relu_config10_s</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>896</td>
</tr>
<tr>
<td>relu_ap_fixed_ap_fixed_16_6_5_3_0_relu_config4_s</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1792</td>
</tr>
<tr>
<td>relu_ap_fixed_ap_fixed_16_6_5_3_0_relu_config7_s</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>896</td>
</tr>
</tbody>
</table>

**Reuse factor = 1**

7 | 0 | 39202448478485 | 0 |

<table>
<thead>
<tr>
<th>Module</th>
<th>BRAM_18KDSP48E</th>
<th>FF</th>
<th>LUT</th>
<th>URAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>dense_latency_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_1</td>
<td>0</td>
<td>12813415</td>
<td>55546</td>
<td>0</td>
</tr>
<tr>
<td>dense_latency_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_2</td>
<td>0</td>
<td>6410839</td>
<td>28307</td>
<td>0</td>
</tr>
<tr>
<td>dense_latency_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_s</td>
<td>0</td>
<td>6410207</td>
<td>27310</td>
<td>0</td>
</tr>
<tr>
<td>dense_latency_ap_fixed_ap_fixed_config11_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0</td>
<td>0</td>
<td>10</td>
<td>2409</td>
<td>4541</td>
</tr>
<tr>
<td>relu_ap_fixed_ap_fixed_16_6_5_3_0_relu_config10_s</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>896</td>
</tr>
<tr>
<td>relu_ap_fixed_ap_fixed_16_6_5_3_0_relu_config4_s</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1792</td>
</tr>
<tr>
<td>relu_ap_fixed_ap_fixed_16_6_5_3_0_relu_config7_s</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>896</td>
</tr>
<tr>
<td>softmax_latency_ap_fixed_ap_fixed_softmax_config13_s</td>
<td>4</td>
<td>5</td>
<td>93</td>
<td>78</td>
</tr>
</tbody>
</table>

**Reuse factor = 16**

8 | 4 | 27136963119366 | 0 |

**Reuse factor 1 → 16**

93% DSP reduction

50% FF increase

52% LUT increase

DSP is usually scarce in FPGA (e.g. total 5952 in U50) compared to FF (1,743K) and LUT (872K)
Compare the Latency between Different Reuse Factor

<table>
<thead>
<tr>
<th>Module</th>
<th>Latency (cycles)</th>
<th>Latency (absolute)</th>
<th>Interval (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>dense_latency_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_1</td>
<td>1</td>
<td>1</td>
<td>5.000 ns</td>
</tr>
<tr>
<td>dense_latency_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_s</td>
<td>1</td>
<td>1</td>
<td>5.000 ns</td>
</tr>
<tr>
<td>dense_latency_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_2</td>
<td>1</td>
<td>1</td>
<td>5.000 ns</td>
</tr>
<tr>
<td>dense_latency_ap_fixed_ap_fixed_config11_0_0_0_0_0_0</td>
<td>1</td>
<td>1</td>
<td>5.000 ns</td>
</tr>
<tr>
<td>relu_ap_fixed_ap_fixed_16_6_5_3_0_relu_config4_s</td>
<td>0</td>
<td>0</td>
<td>0 ns</td>
</tr>
<tr>
<td>relu_ap_fixed_ap_fixed_16_6_5_3_0_relu_config7_s</td>
<td>0</td>
<td>0</td>
<td>0 ns</td>
</tr>
<tr>
<td>relu_ap_fixed_ap_fixed_16_6_5_3_0_relu_config10_s</td>
<td>0</td>
<td>0</td>
<td>0 ns</td>
</tr>
</tbody>
</table>

Reuse factor = 1

Total Latency increases by 7.42 X

<table>
<thead>
<tr>
<th>Module</th>
<th>Latency (cycles)</th>
<th>Latency (absolute)</th>
<th>Interval (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>dense_latency_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_1</td>
<td>9</td>
<td>945.000 ns</td>
<td>45.000 ns</td>
</tr>
<tr>
<td>dense_latency_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_2</td>
<td>12</td>
<td>1260.000 ns</td>
<td>60.000 ns</td>
</tr>
<tr>
<td>dense_latency_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_2</td>
<td>10</td>
<td>1050.000 ns</td>
<td>50.000 ns</td>
</tr>
<tr>
<td>dense_latency_ap_fixed_ap_fixed_config11_0_0_0_0_0_0</td>
<td>13</td>
<td>1365.000 ns</td>
<td>65.000 ns</td>
</tr>
<tr>
<td>relu_ap_fixed_ap_fixed_16_6_5_3_0_relu_config4_s</td>
<td>0</td>
<td>0</td>
<td>0 ns</td>
</tr>
<tr>
<td>relu_ap_fixed_ap_fixed_16_6_5_3_0_relu_config7_s</td>
<td>0</td>
<td>0</td>
<td>0 ns</td>
</tr>
<tr>
<td>relu_ap_fixed_ap_fixed_16_6_5_3_0_relu_config10_s</td>
<td>0</td>
<td>0</td>
<td>0 ns</td>
</tr>
<tr>
<td>softmax_latency_ap_fixed_ap_fixed_softmax_config13_s</td>
<td>2</td>
<td>210.000 ns</td>
<td>10.000 ns</td>
</tr>
</tbody>
</table>

Reuse factor = 16
Long Synthesis Time for gnn_simple

1. Set vertex size to 10 & edge size to 20
2. We change at “parameters.h”
Next Steps

- System performance profiling
- Explore/optimize HLS flow and critical kernels
Next Steps

1. Get the performance at system level (including memory access)
   a. Vitis provides performance from system level view
   b. Try on U50 FPGA

2. Explore the design space of generating template HLS code
   a. Explore/optimize nnet_graph.h (corresponding functions to each NN layer)
   b. Explore/optimize pipeline architectures
Questions
Question 1: Miss match in *reuse_factor* in the code

According to the paper:

In *hls4ml*, this is configured with a “reuse factor” that sets the number of times a multiplier is used in the computation of a layer’s output values. With a reuse factor of one, the computation is fully parallel, i.e., each multiplier is used once. With a reuse factor of $R$, $1/R$ of the computation is done at a time with a factor of $1/R$ fewer multipliers. To make routing more convenient, often there are preferred values of $R$ depending on the dimensions of the matrix itself.

We see the use of *reuse_factor* in the code:

```cpp
#pragma HLS PIPELINE II=CONFIG_T:::reuse_factor
```

According to the SDAccel documentation:

- **II** = <int>: Specifies the desired initiation interval
  
  A pipelined function or loop can process new inputs every <N> clock cycles, where <N> is the II of the loop or function.

We see HLS PIPELINE pragma’s without *reuse_factor* set as its II argument in some parts of the code.

Isn’t the delay caused by upstream initiation suppose to affect the initiations of downstream pipelines?
Question 2: Main Issue of Porting to Vitis?

- Doesn’t Vitis and Vivado share the same backend?
- Vitis provides more system-level view and analysis tools

Building a project with Xilinx Vivado HLS (after downloading and installing from here)

Note: Vitis HLS is not yet supported. Vivado HLS versions between 2018.2 and 2020.1 are recommended.

```python
# Use Vivado HLS to synthesize the model
# This might take several minutes
hls_model.build()

# Print out the report if you want
hls4ml.report.read_vivado_report('my-hls-test')
```
Question 3: Suggestions on Further Optimization?

- C++ code is currently used for datapath modules? (not much of control?)
- Any suggestions to modify/optimize these C++ codes?
- Which functions are more critical to users? to explore first.
Thank you!