CGRA Mapping Using Zero-Suppressed Binary Decision Diagrams

Rami Beidas and Jason Anderson
University of Toronto

27th Asia and South Pacific Design Automation Conference
January 20, 2022
CGRA: Coarse Grained Reconfigurable Architecture

- Array of programmable processing elements (PEs)
- PEs are word-level functional unit (think ALU)
- PEs are connected to nearest neighbours through word-level programmable switches arranged in a regular topology, like a mesh or torus
- Less silicon committed to programmability
- Lie between ASICs and FPGAs on the spectrum of power, performance, area, and flexibility
CGRA Mapping

- The key CAD step for implementing an application on a CGRA
- Inputs are application kernel compiled into a dataflow graph (DFG) and a device model graph
- Mapping assigns each DFG node to a device vertex and each DFG edge to a set of device arcs

```c
foo() {
    a = ...;
    b = ...;
    c = 5 * a + b;
    ... = c ;
}
```
The Challenge

- CGRA routing is highly restricted when compared to modern FPGAs
- This restriction limits the success of traditional CAD solution
- Difficult to decouple placement and routing
- Many heuristic solutions were proposed, including simulated annealing [1], genetic algorithms [2], and graph embedding [3] among others; many of which are architecture-specific
- Some have shown such heuristics can be ineffective for highly constrained problems and opted for optimal or near optimal solutions using ILP formulation and general optimization solvers [4][5]
- Unfortunately, such solutions with general solvers have not been shown to scale

Zero Suppressed Binary Decision Diagrams (ZDDs)

- A compact representation for solving problems in set theory [1]
- A ZDD represents a family of sets as a DAG, with internal nodes representing elements that appear in at least one set, and two terminal nodes $\bot$ and $T$
- Every internal node has HI/LO edges pointing to the residual subfamilies that do/do not contain the source element
- Paths from the root node to $T$ represent the family members

Zero Suppressed Binary Decision Diagrams (ZDDs)

- Set operations on ZDD are implemented using recursive procedures that utilize dynamic programming [1][2]
- Efficient implementations available for set union, intersection, difference, product, maximal, minimal, subset, superset, ... etc.
- Utilized in a variety of applications, including logic synthesis, graph optimization, and data mining among others
- Then... a hibernation!

Simple Path Enumeration and SIMPATH

- ZDD was proposed as an efficient representation for enumeration of simple (cycle free) paths in undirected graphs, along with fast algorithm for the ZDD construction called SIMPATH [1]
- For an 8x8 mesh, ~800 billion paths were represented using ~33K node ZDD
- The proposed solution reignited research in ZDD applications, especially in graph enumerations [2]

Intuition

- You might start to see how the path enumeration is related to our mapping problem
- A single DFG node mapping is simply a set of edges from where the node is mapped to where all the uses are mapped
- Each mapping solution is simply a set of used edges in the device annotated by owning DFG value
Problem Formulation

- The input
  - Application kernel DFG with operation nodes \( N \) and dataflow edges \( E \subseteq N \times N \)
  - Device model graph with vertices \( V \) representing PEs and arcs \( A \subseteq V \times V \) representing routing
  - The set of opcodes
    \( O = \{ \text{IN}, \text{OUT}, \text{LD}, \text{STR}, \text{ADD}, \text{SUB}, \ldots \} \)
    - \( OP : N \rightarrow O \)
    - \( OPS : V \rightarrow P(O) \)

- The output
  - Mapping \( N \rightarrow V \) and \( E \rightarrow P(A) \)
Problem Encoding - Device Domains

We define three discrete domains of ZDD variables to represent device entities:

- \( W \), which corresponds to the set of all device arcs or interconnects
- \( D \), which corresponds to the set of all device vertices as path sources such that \( d_v \) implies a route \textit{from} \( v \)
- \( S \), which corresponds to the set of all device vertices as path sinks such that \( s_v \) implies a route \textit{to} \( v \)
We developed a simple path enumeration solution for directed graphs, constrained by hop count.

Given the nature of the problem, the solution is faster and simpler than SIMPATH.

Returns a table of ZDDs, one for each device vertex, summarizing all paths starting at that vertex.

\[ V \mapsto \text{Set of Simple Paths (Zdd)} \]
CGRA Path Enumeration

- For hop count $h = 2$:
  $SpZdd[0] = \{d_0, w_0, s_2\}, \{d_0, w_0, w_2, s_3\}, \{d_0, w_0, w_5, s_4\}$,
  $SpZdd[1] = \{d_1, w_1, s_2, w_3\}, \{d_1, w_1, s_3\}, \{d_1, w_1, w_7, s_5\}$, ... etc
  the total number of paths is 28
- For $h = 3$, the total number of paths is 46
- For $h = 4$, the count increases to 58
- For $h = 5$, it becomes 60
- No paths have more than 5 hops
Problem Encoding - DFG Domains

To represent DFG mappings, we define another set of ZDD variable domains:

- $W'$, which corresponds to the set of all interconnects in the device, with one-to-one mapping to $W$
- $D'$, which corresponds to the set of all possible mappings of DFG source nodes to device vertices such that $d'_{v,n}$ implies a dataflow edge from node $n$ mapped to vertex $v$
- $S'$, which corresponds to the set of all possible mappings of DFG sink nodes to device vertices such that $s'_{v,n}$ implies a dataflow edge to node $n$ mapped to $v$
Single DFG Node Mapping Enumeration

- If we take a single DFG node in isolation and consider mapping it to a device vertex, we can enumerate all possible mappings of the dataflows to the fanout of that node.
- In a nutshell, the fanout of a node \( n \) mapped to vertex \( v \) is the cartesian product of all possible routes to all possible placements of \( n \)'s fanouts, performed in the DFG domains.

```plaintext
Algorithm 2 Node Mapping Enumeration
1: function ENUMNODEMAP(n, v)
2:     \( M = \{ m \in N : (n, m) \in E \} \)
3:     if \( |M| = 0 \) then
4:         \( rZdd = \{ d'_{v,n} \} \)
5:     else
6:         \( rZdd = \top \)
7:         for all \( m \in M \) do
8:             \( spZdd = \text{REN}(SpZdd[v], D, S) \)
9:             \( rZdd = \text{CARTPROD}(rZdd, spZdd) \)
10:        \( rZdd = \text{LEGA}(rZdd, M) \)
11:    return rZdd
```
Single DFG Node Mapping Enumeration

- Enumerating all possible mappings of \( n = 2 \) to \( v = 2 \), while restricting hop count to 2, yields three possible mappings:
  \[
  \begin{align*}
  &\{\{w_2, d'_{2,2}, w_7, s'_{5,4}\}, \\
  &\{w_2, d'_{2,2}, s'_{3,4}\}, \\
  &\{w_5, d'_{2,2}, w_8, s'_{5,4}\}\}
  \end{align*}
  \]
DFG Mapping Enumeration

- Using single node mapping enumeration, we can enumerate the entire DFG mappings basically as a cartesian product of all DFG nodes’ mappings
- Legalization steps drop solutions that overuse resources
DFG Mapping Enumeration

- With I/Os pinned to simplify the example, mapping enumeration returns three solutions:
  \[
  \{w'_0, d'_0, w'_1, d'_1, w'_2, d'_2, w'_7, d'_3, w'_{11}, d'_{7,5}\},
  \{w'_0, d'_0, w'_1, d'_1, w'_5, d'_2, w'_7, w'_8, w'_{11}, d'_{5,4}, d'_{7,5}\},
  \{w'_0, d'_0, w'_1, d'_1, w'_5, w'_7, w'_8, d'_{4,2}, w'_{11}, d'_{5,4}, d'_{7,5}\}
  \]
- The ZDD representing all possible solutions is a DAG, choosing an optimal solution is as simple as assigning cost to ZDD variables and running linear time shortest path from root to \( T \)
- Minimizing routing yields the first mapping
Detailed DFG Mapping Enumeration

- Note that DFG nodes mapping to PEs is explicit, but the exact value to wire mapping is implicit; another run of the algorithm, but with wire domain $W''$ annotated with DFG nodes, such that $w''_{a,n}$ implies arc a caries value produced by DFG node $n$
- Breaking the problem in two steps allows us to use $O(E)$ ZDD variables for interconnects domain in the larger problem instead of $O(E \times N)$
Runtime Control

- Even with a highly efficient data structure to represent all possible mappings, the number of solutions is still massive and the size of the enumeration ZDD still explodes for larger problems.
- Most of the enumerated solutions are far from optimal.
- Therefore, we relied on two techniques to keep runtime in check:
  - Pre-Placement
  - Iterative Minimum
- Experimentally, these techniques have minimal impact on quality of results:
  - In most cases an optimal solution is found
  - In fewer cases the solution is just few interconnects away from optimal (<5%)
Pre-Placement

- The idea is to have an optional placement step using traditional solution such as simulated annealing to limit the enumeration space of valid solutions
- In case the placement result is too restrictive, we still allow a user defined tolerance to help the routing step
Iterative Minimum

- Many of the enumerated partial mappings are far from optimal
- With iterative minimum, with each iteration we only keep minimum cost partial mappings
- The MIN function returns all minimum cost sets in a single pass
- Iteration count is user defined

Algorithm 4 Iterative Minimum

```
1: function ITERMIN(sZdd, MIC)
2:   rZdd = ⊥
3:   for i ∈ [0, MIC − 1] do
4:     minZdd = MIN(sZdd)
5:     rZdd = UNION(rZdd, minZdd)
6:   sZdd = DIFF(sZdd, minZdd)
7:   return rZdd
```
Experimental Study

- We use LLVM compiled kernels from benchmarks distributed with CGRA-ME.
- We target a single-context HyCube.
- We compare our mapper with optimal and heuristic mappers of the current CGRA-ME release.
- Two orders of magnitude speedup was obtained.

<table>
<thead>
<tr>
<th>Kernel Name</th>
<th>DFG Size</th>
<th>CGRA Size</th>
<th>ILP [4] Runtime(s)</th>
<th>Heu [5] Runtime(s)</th>
<th>This Work Runtime(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>accumulate</td>
<td>18</td>
<td>4x4</td>
<td>231.83</td>
<td>TO</td>
<td>0.23</td>
</tr>
<tr>
<td>cap</td>
<td>24</td>
<td>6x6</td>
<td>1881.81</td>
<td>65.27</td>
<td>0.39</td>
</tr>
<tr>
<td>conv3</td>
<td>16</td>
<td>4x4</td>
<td>11.82</td>
<td>11.86</td>
<td>0.18</td>
</tr>
<tr>
<td>conv2</td>
<td>24</td>
<td>6x6</td>
<td>132.72</td>
<td>63.94</td>
<td>0.31</td>
</tr>
<tr>
<td>mac2</td>
<td>24</td>
<td>6x6</td>
<td>TO</td>
<td>TO</td>
<td>0.29</td>
</tr>
<tr>
<td>matrixmult</td>
<td>17</td>
<td>4x4</td>
<td>7.56</td>
<td>25.78</td>
<td>0.18</td>
</tr>
<tr>
<td>mults2</td>
<td>25</td>
<td>6x6</td>
<td>2935.43</td>
<td>108.86</td>
<td>2.83</td>
</tr>
<tr>
<td>nomem1</td>
<td>6</td>
<td>4x4</td>
<td>4.27</td>
<td>4.11</td>
<td>0.10</td>
</tr>
<tr>
<td>simple2</td>
<td>12</td>
<td>6x6</td>
<td>43.25</td>
<td>93.78</td>
<td>0.32</td>
</tr>
<tr>
<td>simple2</td>
<td>12</td>
<td>4x4</td>
<td>57.99</td>
<td>19.77</td>
<td>0.29</td>
</tr>
<tr>
<td>sum</td>
<td>7</td>
<td>4x4</td>
<td>2.36</td>
<td>11.48</td>
<td>0.11</td>
</tr>
</tbody>
</table>

Experimental Study

- Larger problems beyond the capability of previous solutions were also evaluated varying parameters of the runtime control techniques
- In general, increasing tolerance and iteration count increases the number of enumerated solutions, possibly from 0
- It is possible for a pre-placement to be infeasible to route; hence, the need for increasing tolerance
- Runtime can grow exponentially with higher iteration counts and, more severely, pre-placement tolerance; therefore, use must be with caution
Conclusion and Future Work

- We presented a ZDD-based CGRA mapper and illustrated its speed advantage when compared to state-of-the-art exact and heuristic solvers.
- The immediate next step would be to support:
  - multi-context CGRA architectures
  - multi-output operations
  - predicated execution
- We believe our solution is flexible enough to support these features systematically without sacrificing speed or quality of results.
- The next major development would utilize the enumeration feature of our solution to guide the design of domain-specific CGRA architectures.
Thank You for Listening, Questions?