CGRA Mapping Using Zero-Suppressed Binary Decision Diagrams

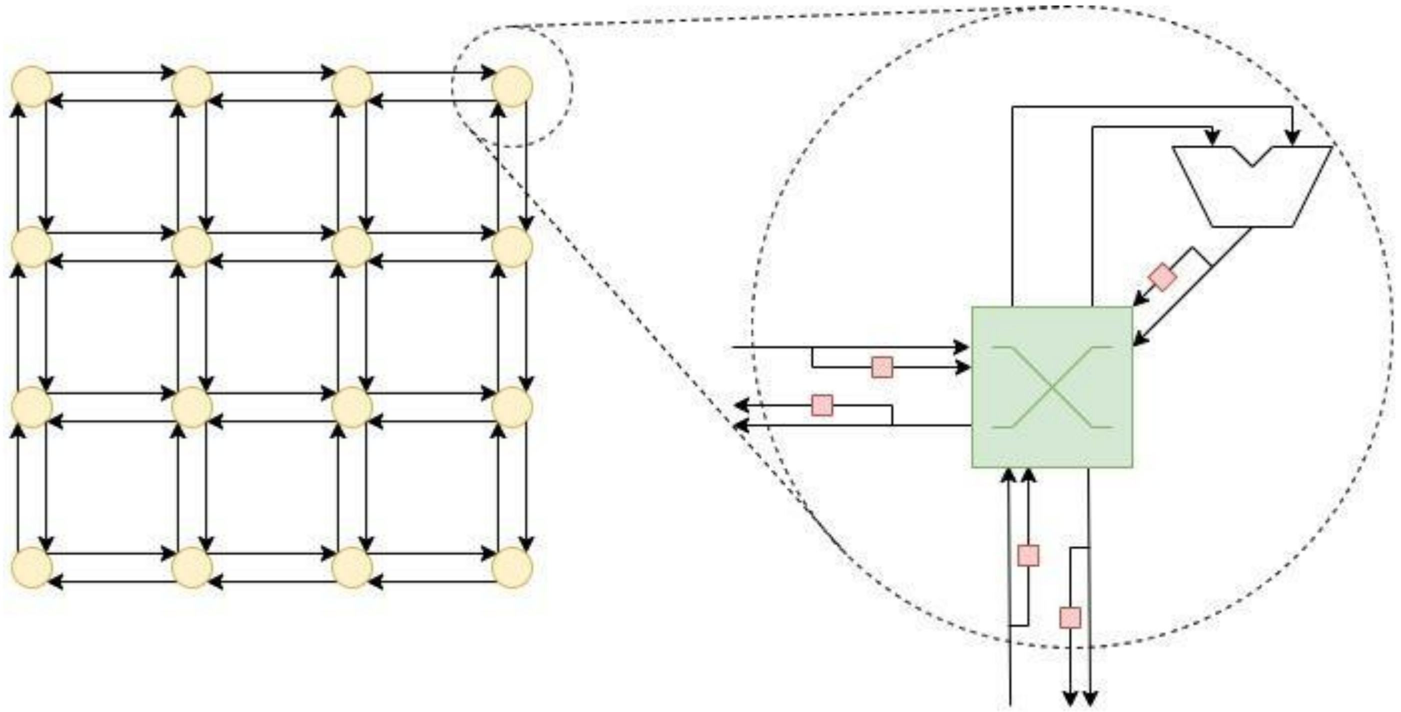
Rami Beidas and Jason Anderson
University of Toronto

27th Asia and South Pacific Design Automation Conference
January 20, 2022

The Edward S. Rogers Sr. Department of Electrical & Computer Engineering
UNIVERSITY OF TORONTO

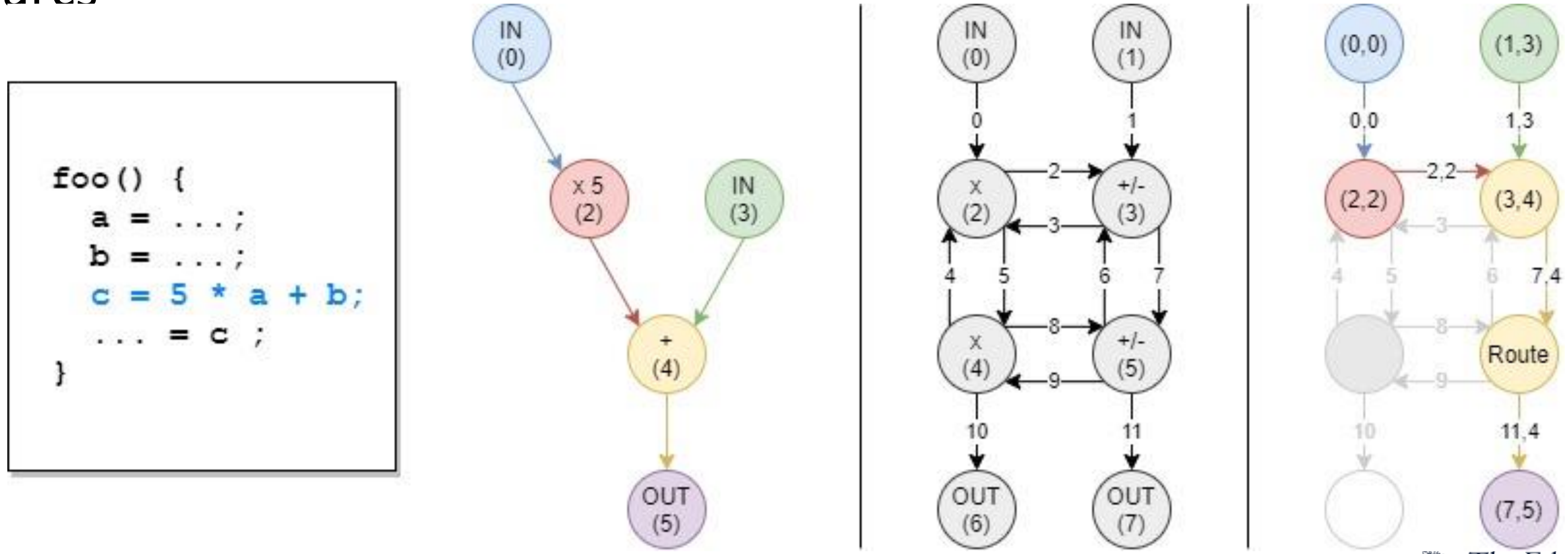
CGRA: Coarse Grained Reconfigurable Architecture

- 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



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

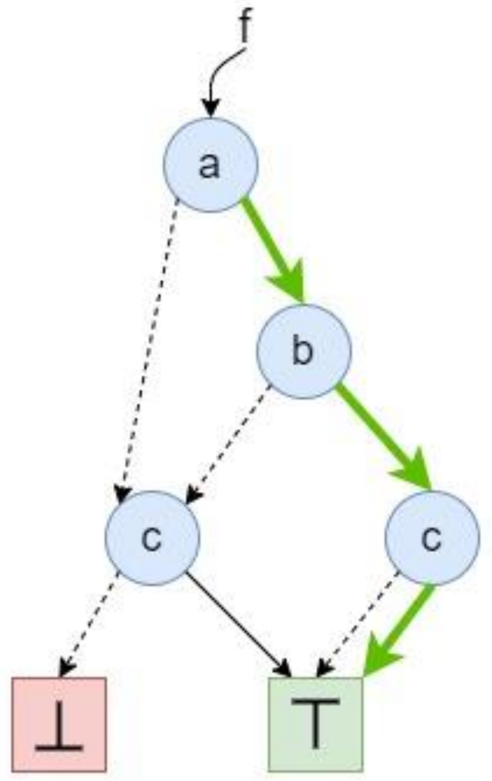
The Challenge

- 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

[1] B. Mei, et al, "DRESC: a retargetable compiler for coarse-grained reconfigurable architectures," in Proc. FPT, 2002.
[2] T. Kojima, et al, "GenMap: A genetic algorithmic approach for optimizing spatial mapping of coarse-grained reconfigurable architectures," IEEE TVLSI, vol. 28, no. 11, 2020.
[3] L. Chen and T. Mitra, "Graph minor approach for application mapping on CGRAs," in Proc. FPT, 2012.
[4] S. Chin and J. Anderson, "An architecture-agnostic integer linear programming approach to CGRA mapping," in Proc. DAC, 2018.
[5] M. J. P. Walker and J. Anderson, "Generic connectivity-based CGRA mapping via integer linear programming," in IEEE FCCM, 2019.

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 ⊥ 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



f = {{c},{a,c},{a,b},**{a,b,c}**}

[1] S. Minato, “Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems,” in IEEE/ACM DAC, 1993

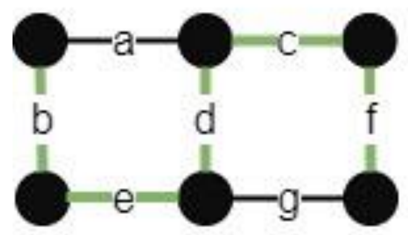
Zero Suppressed Binary Decision Diagrams (ZDDs)

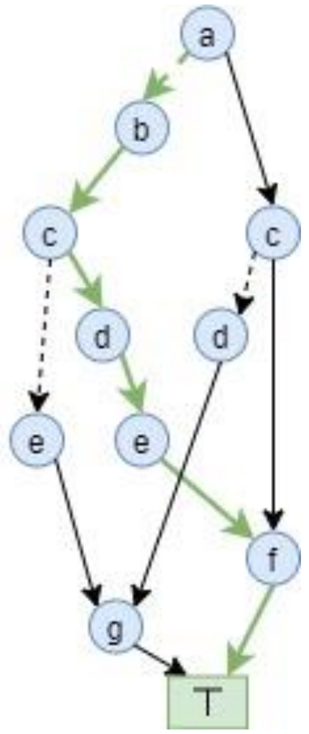
- Set operations on ZDD are implemented using recursive procedures that utilize dynamic programming [1][2]
- Efficient implementations available for set union, intersection, difference, product, maximal, minimal, subset, superset, … etc.
- Utilized in a variety of applications, including logic synthesis,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

[1] S. Minato, “Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems,” in IEEE/ACM DAC, 1993
[2] A. Mishchenko, “An Introduction to Zero-Suppressed Binary Decision Diagrams,” Portland SU, Tech. Rep., 2001.

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]

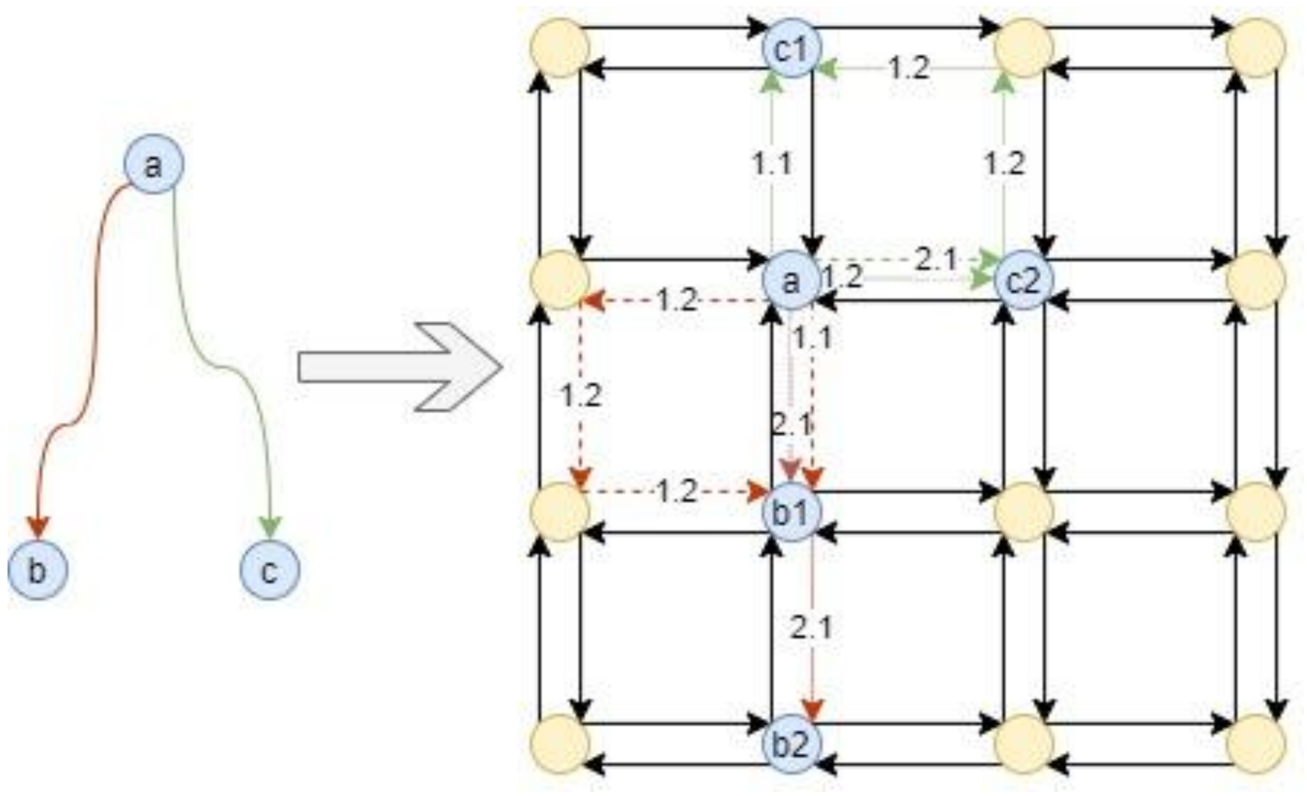




[1] D. E. Knuth, "The Art of Computer Programming", Addison-Wesley, 2011, vol. 4A: Combinatorial Algorithms
[2] H. Iwashita, et al, "ZDD-Based Computation of the Number of Paths in a Graph", TCS-TR-A-12-60, September 18, 2012

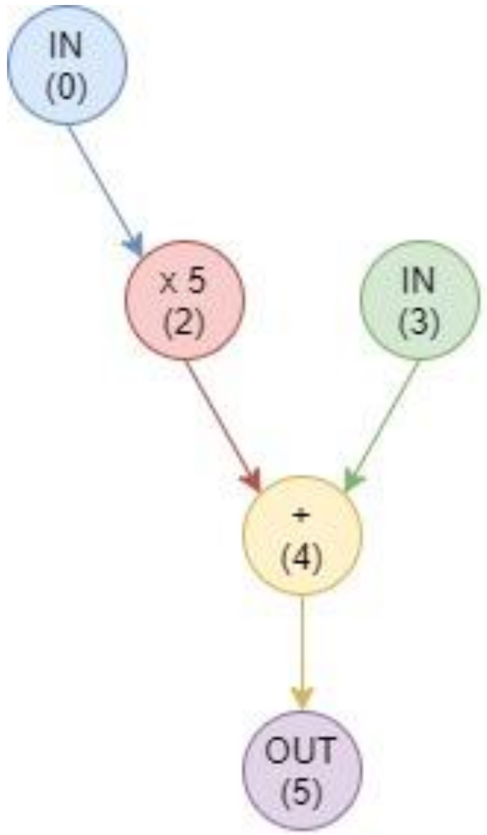
Intuition

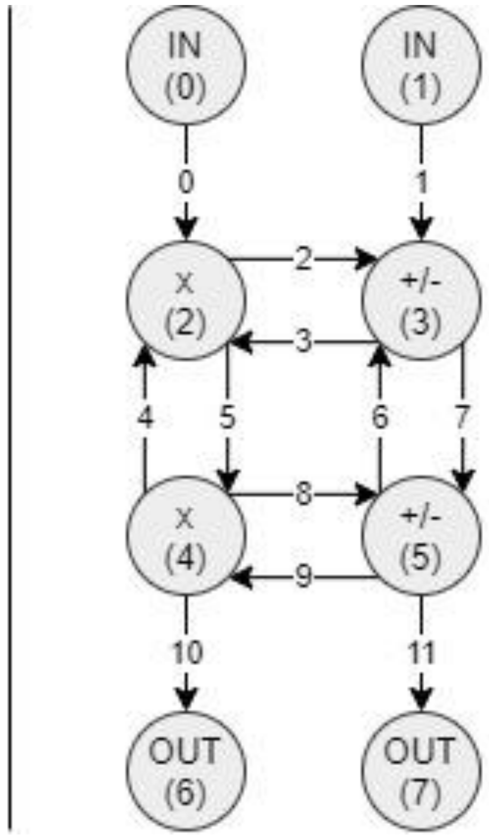
- You might start to see how the path enumeration is related to our mapping problem
- A single DFG node mapping is simply a set of edges from where the node is mapped to where all the uses are mapped
- Each mapping solution is simply a set of used edges in the device annotated by owning DFG value



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 $O = \{IN, OUT, LD, STR, ADD, SUB, \ldots\}$
    - $OP: N \rightarrow O$
    - $OPS: V \rightarrow \boldsymbol{P}(O)$
- The output
  - Mapping $N \rightarrow V$ and $E \rightarrow \boldsymbol{P}(A)$





Problem Encoding - Device Domains

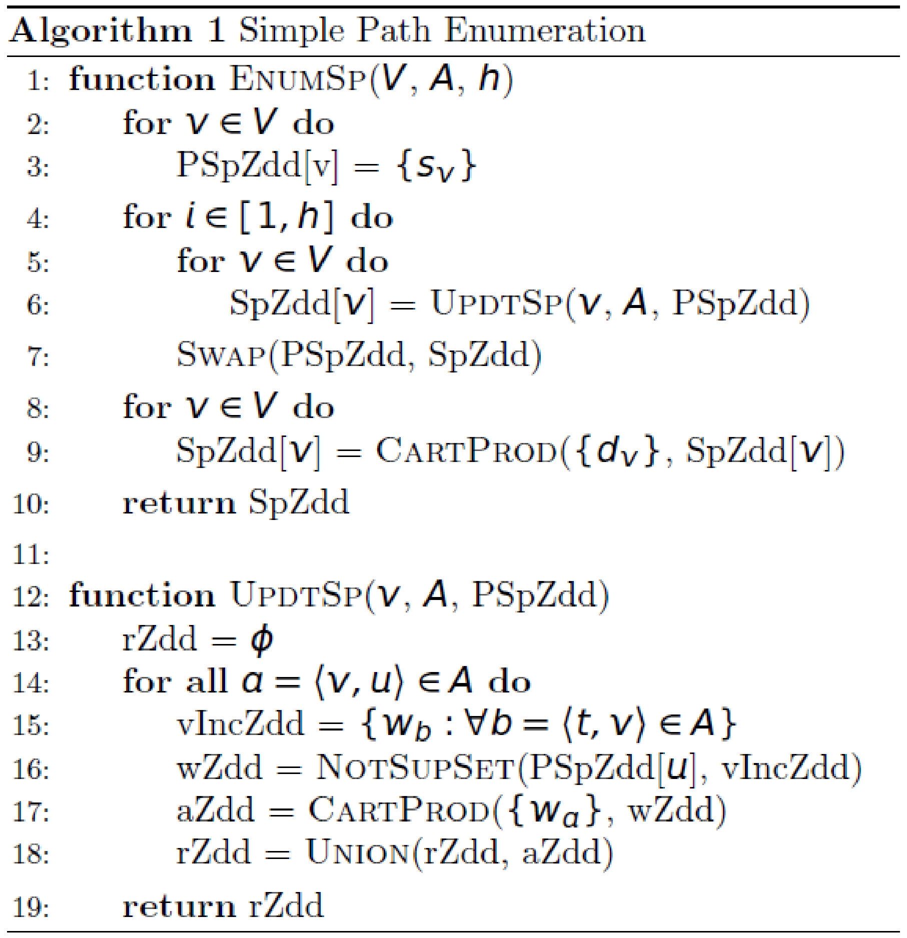
We define three discrete domains of ZDD variables to represent device entities:

- 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 *from* v
- S, which corresponds to the set of all device vertices as path sinks such that s_v implies a route *to* v

CGRA Path Enumeration

- 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

  $V \mapsto$ Set of Simple Paths (Zdd)



Algorithm 1 Simple Path Enumeration

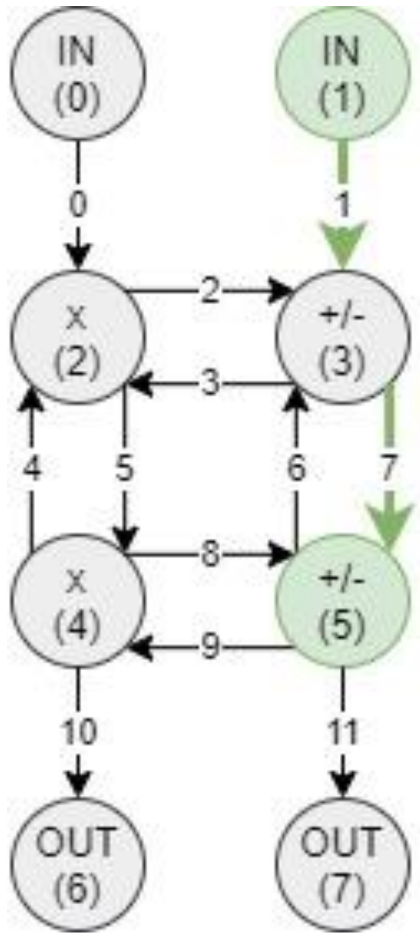
```
function EnumSp($V$, $A$, $h$)
    for $v \in V$ do
        PSpZdd[v] = $\{s_v\}$
    for $i \in [1, h]$ do
        for $v \in V$ do
            SpZdd[$v$] = UpdtSp($v$, $A$, PSpZdd)
        Swap(PSpZdd, SpZdd)
    for $v \in V$ do
        SpZdd[$v$] = CartProd($\{d_v\}$, SpZdd[$v$])
    return SpZdd

function UpdtSp($v$, $A$, PSpZdd)
    rZdd = $\phi$
    for all $a = \langle v, u \rangle \in A$ do
        vIncZdd = $\{w_b : \forall b = \langle t, v \rangle \in A\}$
        wZdd = NotSupSet(PSpZdd[$u$], vIncZdd)
        aZdd = CartProd($\{w_a\}$, wZdd)
        rZdd = Union(rZdd, aZdd)
    return rZdd
```

The Edward S. Rogers Sr. Department of Electrical & Computer Engineering
UNIVERSITY OF TORONTO

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



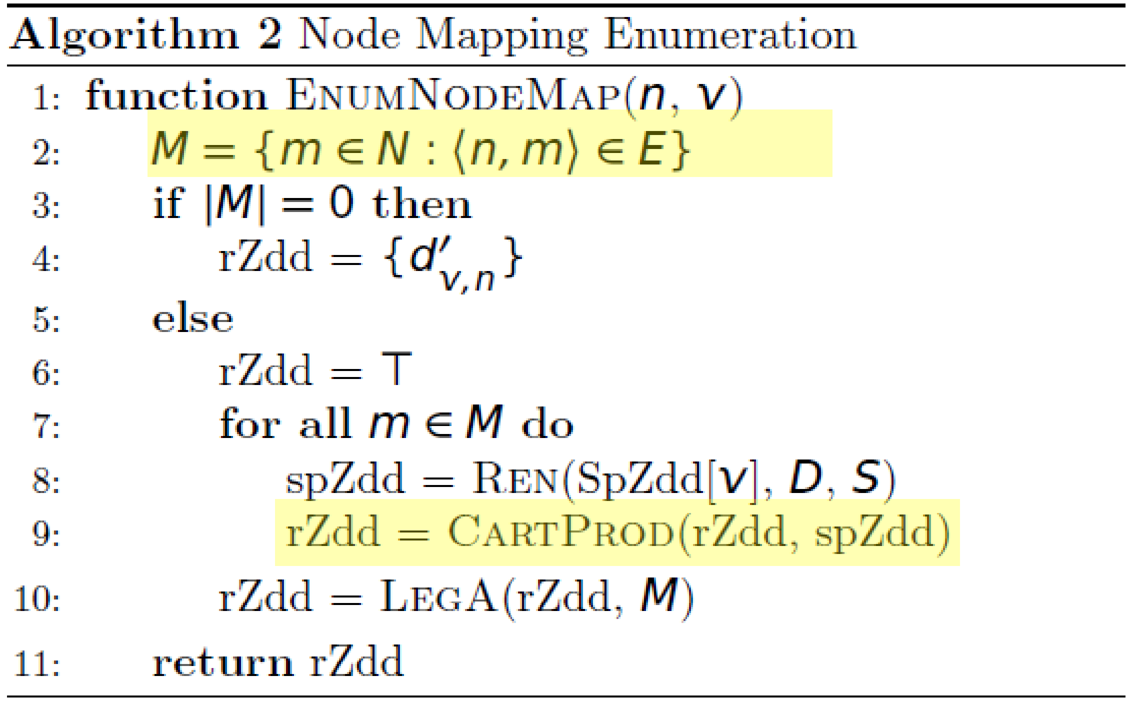
Problem Encoding - DFG Domains

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



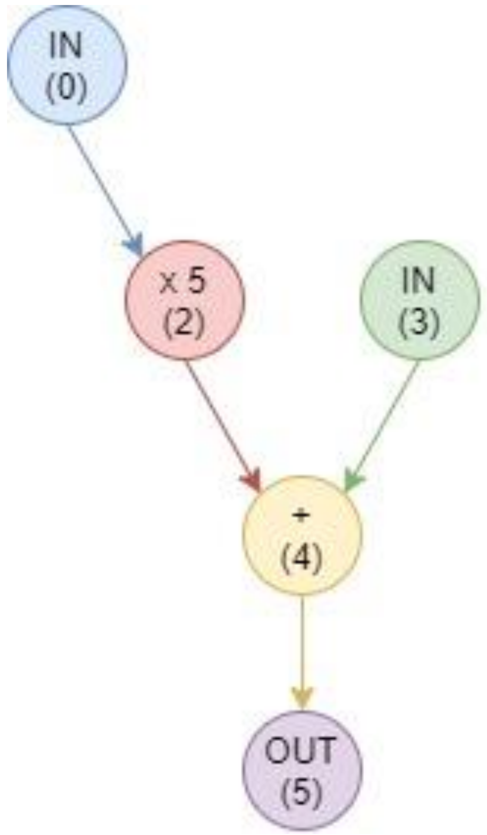
Algorithm 2 Node Mapping Enumeration

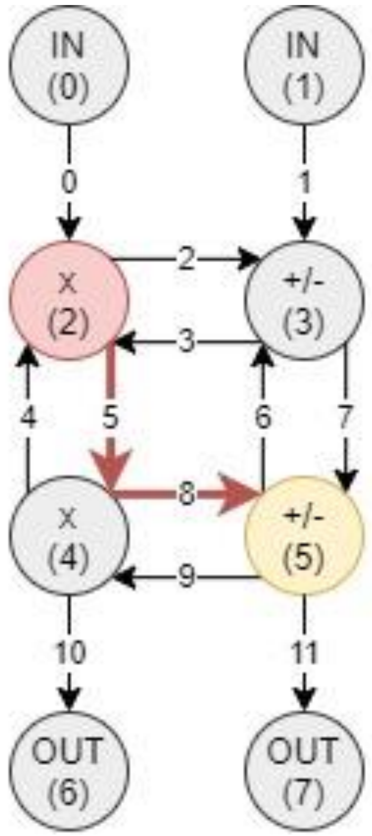
```
function EnumNodeMap($n$, $v$)
    $M = \{m \in N : \langle n, m \rangle \in E\}$
    if $|M| = 0$ then
        rZdd = $\{d'_{v,n}\}$
    else
        rZdd = $\top$
        for all $m \in M$ do
            spZdd = Ren(SpZdd[$v$], $D$, $S$)
            rZdd = CartProd(rZdd, spZdd)
        rZdd = LegA(rZdd, $M$)
    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:

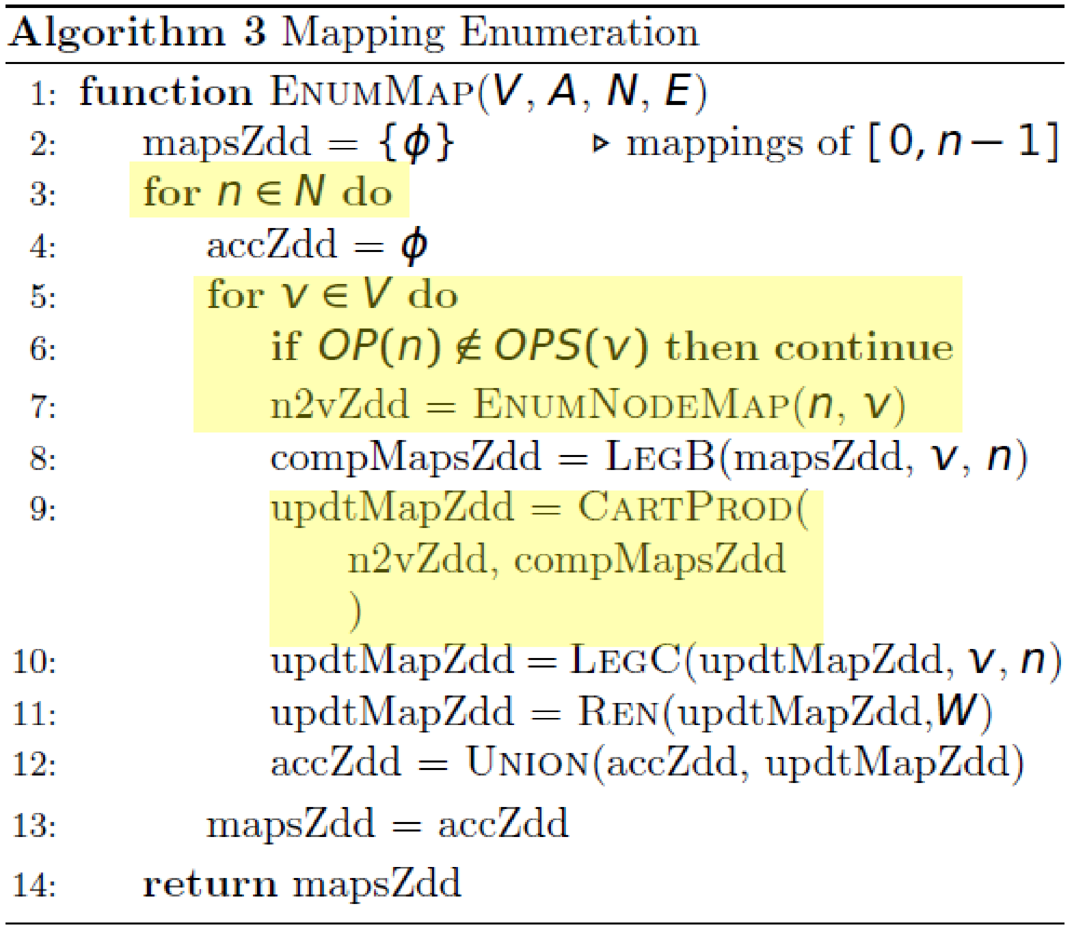
  $\{\{w_2, d'_{2,2}, w_7, s'_{5,4}\},$
  $\{w_2, d'_{2,2}, s'_{3,4}\},$
  $\{w_5, d'_{2,2}, w_8, s'_{5,4}\}\}$





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



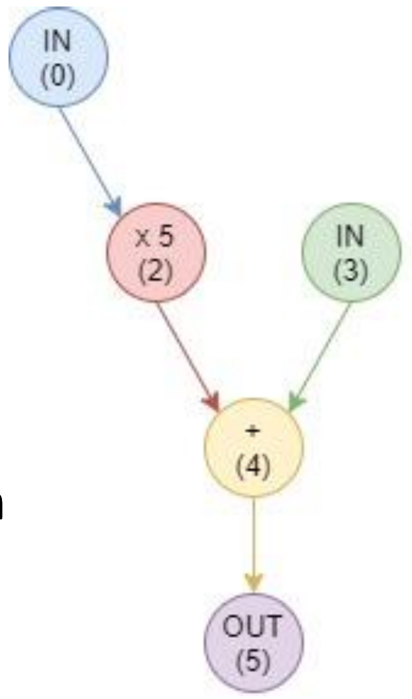
Algorithm 3 Mapping Enumeration

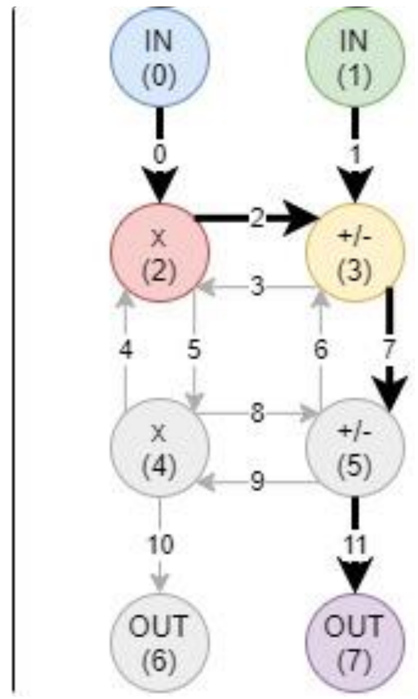
```
function ENUMMAP($V$, $A$, $N$, $E$)
    mapsZdd = $\{\phi\}$                ▷ mappings of $[0, n-1]$
    for $n \in N$ do
        accZdd = $\phi$
        for $v \in V$ do
            if $OP(n) \notin OPS(v)$ then continue
            n2vZdd = ENUMNODEMAP($n$, $v$)
            compMapsZdd = LEGB(mapsZdd, $v$, $n$)
            updtMapZdd = CARTPROD(
                n2vZdd, compMapsZdd
                )
            updtMapZdd = LEGC(updtMapZdd, $v$, $n$)
            updtMapZdd = REN(updtMapZdd,$W$)
            accZdd = UNION(accZdd, updtMapZdd)
        mapsZdd = accZdd
    return mapsZdd
```

DFG Mapping Enumeration

- With I/Os pinned to simplify the example, mapping enumeration returns three solutions:

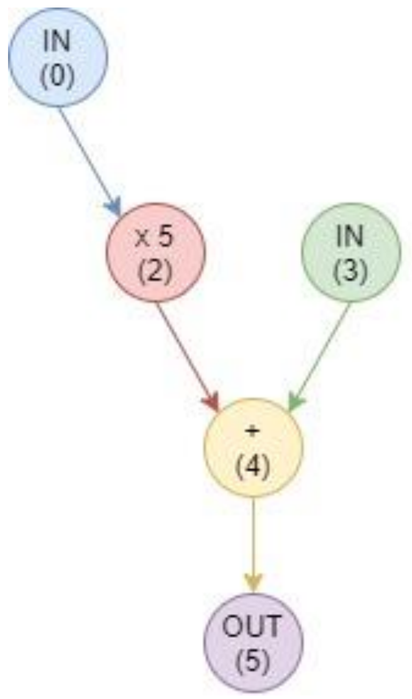
  $\{\{w'_0, d'_{0,0}, w'_1, d'_{1,3}, w'_2, d'_{2,2}, w'_7, d'_{3,4}, w'_{11}, d'_{7,5}\},$
  $\{w'_0, d'_{0,0}, w'_1, d'_{1,3}, w'_5, d'_{2,2}, w'_7, w'_8, w'_{11}, d'_{5,4}, d'_{7,5}\},$
  $\{w'_0, d'_{0,0}, w'_1, d'_{1,3}, w'_5, w'_7, w'_8, d'_{4,2}, w'_{11}, d'_{5,4}, d'_{7,5}\}\}$
- The ZDD representing 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

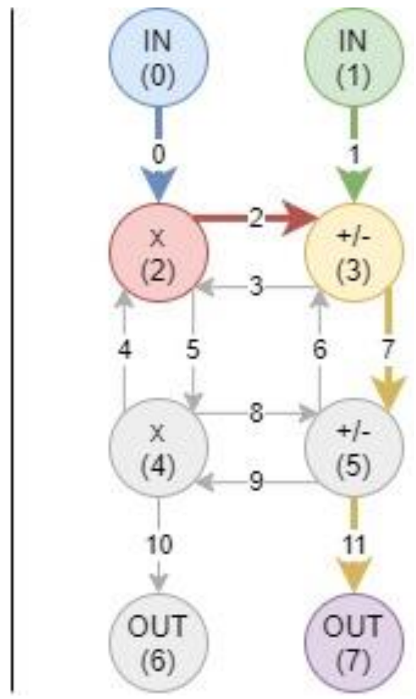




Detailed DFG Mapping Enumeration

- 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 $O(E{\times}N)$



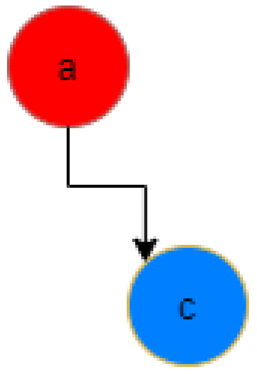


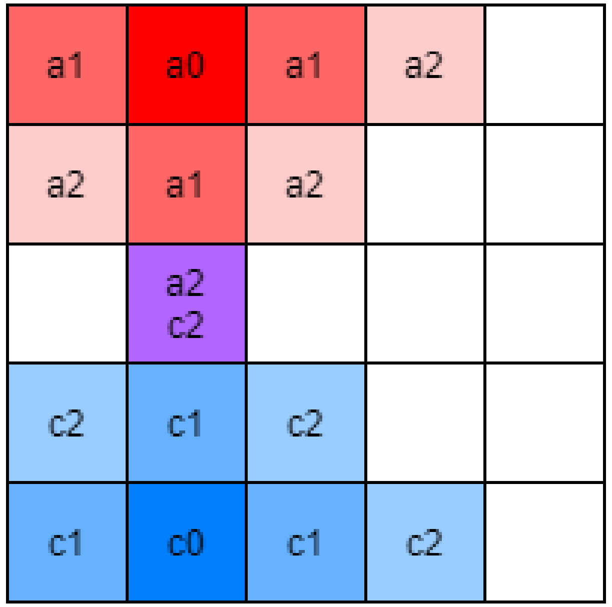
Runtime Control

- 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

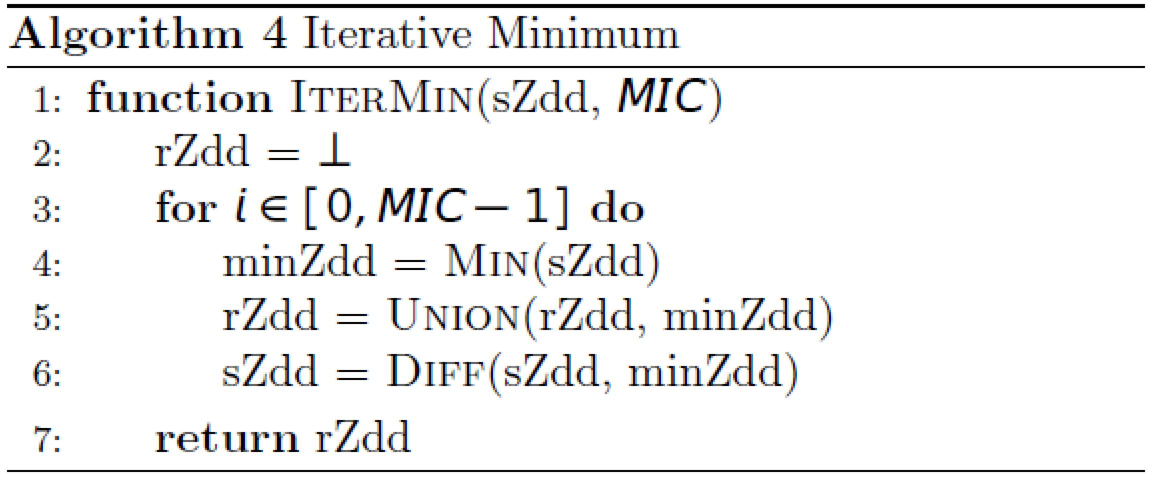




a1	a0	a1	a2	
a2	a1	a2		
	a2 c2			
c2	c1	c2		
c1	c0	c1	c2	

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

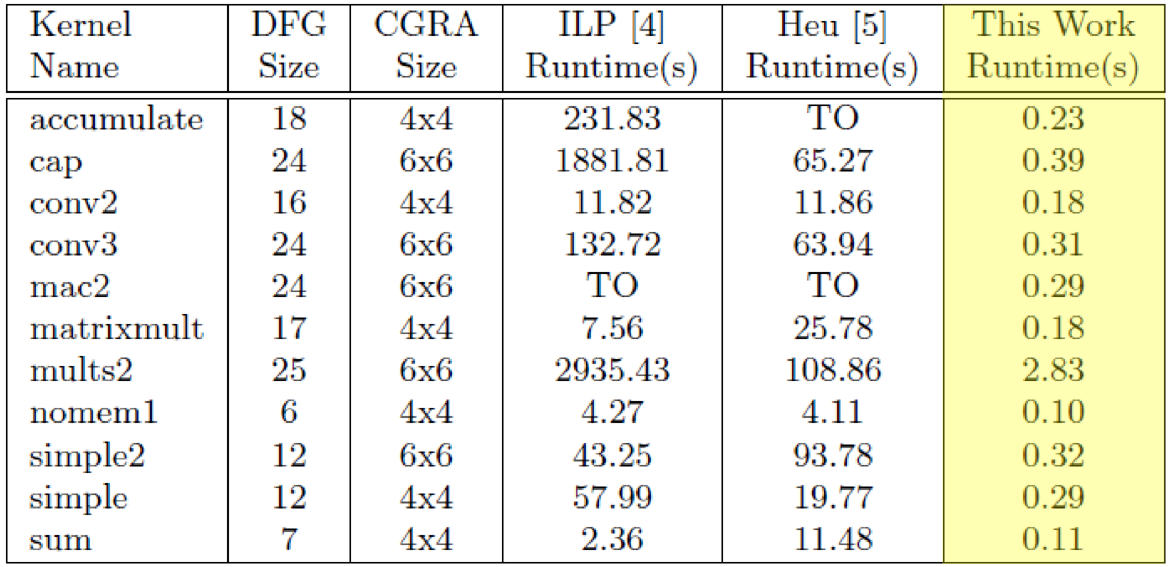


Algorithm 4 Iterative Minimum

```
function ITERMIN(sZdd, $MIC$)
    rZdd $= \perp$
    for $i \in [0, MIC-1]$ do
        minZdd = MIN(sZdd)
        rZdd = UNION(rZdd, minZdd)
        sZdd = DIFF(sZdd, minZdd)
    return rZdd
```

Experimental Study

- The proposed solution was implemented in the CGRA-ME framework [1] utilizing the CUDD [2] and Extra [3] libraries
- 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

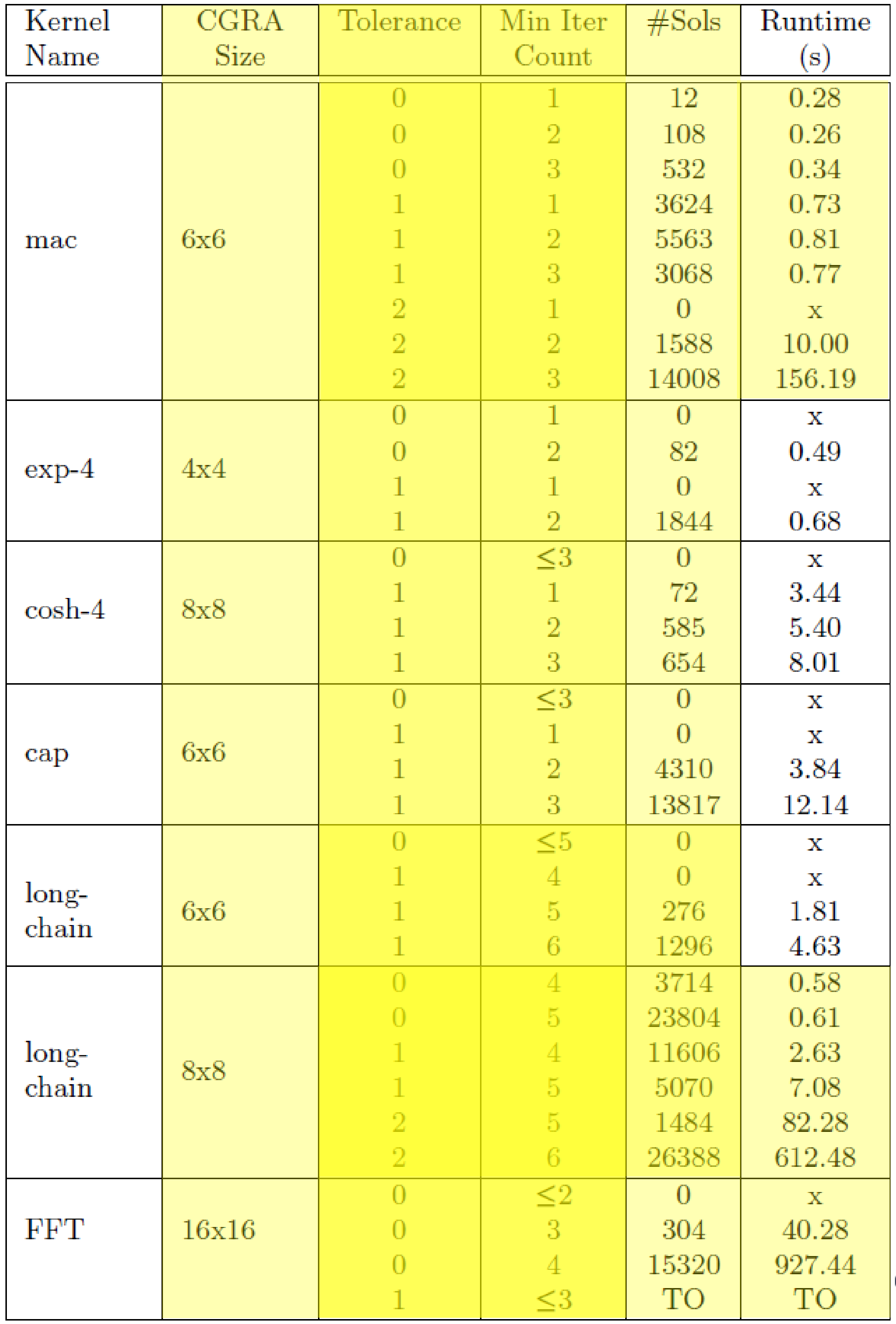


Kernel Name	DFG Size	CGRA Size	ILP [4] Runtime(s)	Heu [5] Runtime(s)	This Work Runtime(s)
accumulate	18	4x4	231.83	TO	0.23
cap	24	6x6	1881.81	65.27	0.39
conv2	16	4x4	11.82	11.86	0.18
conv3	24	6x6	132.72	63.94	0.31
mac2	24	6x6	TO	TO	0.29
matrixmult	17	4x4	7.56	25.78	0.18
mults2	25	6x6	2935.43	108.86	2.83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

[1] J. Anderson et al, "CGRA-ME: An Open-Source Framework for CGRA Architecture and CAD Research", ASAP 2021
[2] F. Somenzi, "CUDD package", Jan 2016. [Online]. Available: https://github.com/ivmai/cudd
[3] A. Mishchenko, "An Introduction to Zero-Suppressed Binary Decision Diagrams," Portland SU, Tech. Rep., June 2001
[4] S. Chin and J. Anderson, "An architecture-agnostic integer linear programming approach to CGRA mapping," in Proc. DAC, 2018.
[5] M. J. P. Walker and J. Anderson, "Generic connectivity-based CGRA mapping via integer linear programming," in IEEE FCCM, 2019.

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



Kernel Name	CGRA Size	Tolerance	Min Iter Count	#Sols	Runtime (s)
mac	6x6	0	1	12	0.28
		0	2	108	0.26
		0	3	532	0.34
		1	1	3624	0.73
		1	2	5563	0.81
		1	3	3068	0.77
		2	1	0	x
		2	2	1588	10.00
		2	3	14008	156.19
exp-4	4x4	0	1	0	x
		0	2	82	0.49
		1	1	0	x
		1	2	1844	0.68
cosh-4	8x8	0	≤3	0	x
		1	1	72	3.44
		1	2	585	5.40
		1	3	654	8.01
cap	6x6	0	≤3	0	x
		1	1	0	x
		1	2	4310	3.84
		1	3	13817	12.14
long-chain	6x6	0	≤5	0	x
		1	4	0	x
		1	5	276	1.81
		1	6	1296	4.63
long-chain	8x8	0	4	3714	0.58
		0	5	23804	0.61
		1	4	11606	2.63
		1	5	5070	7.08
		2	5	1484	82.28
		2	6	26388	612.48
FFT	16x16	0	≤2	0	x
		0	3	304	40.28
		0	4	15320	927.44
		1	≤3	TO	TO

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?