The RAM mapper CAD tool

My implementation of RAM mapper only uses the simple mapping scheme[1], i.e., one type of physical RAM for one logical RAM, no mixed assignment, no physical RAM sharing.

The RAM mapping problem in my implementation is formulated as an Integer Linear Problem (ILP), where each logical RAM has to select one physical RAM among all available physical RAMs for mapping in order to minimize the total area taken by the circuit. The detailed formulation are shown below.

For each logical RAM $L_i \in \mathcal{L},\ |\mathcal{L}| = N_L$, it needs to select one physical RAM $P_j \in \mathcal{P},\ |\mathcal{P}| = N_P$ as its implementation, thus we can use a set of binary decision variables $\mathcal{I}_i$ to encode whether logical RAM L_i choose physical RAM P_j as its implementation. Because there are two types of physical RAMs (LUTRAM and BRAM) and these two types of RAMs have slightly different calculations in the final area equation, I split $\mathcal{I}_i$ into two subsets for LUTRAM and BRAM ($\mathcal{B},\ |\mathcal{B}| = N_B$) respectively. For example, $I_i^{B_j}$ is the L_i's decision variable for j-th BRAM, I_i^{LUTRAM} is the decision variable for LUTRAM (because there are at most one type of LUTRAM in FPGA).

$$\mathcal{I}_i = (I_i^{B_1}, I_i^{B_2}, \ldots, I_i^{B_{N_B}}, I_i^{LUTRAM}) \in \{0,1\}^{N_B+1} \tag{1}$$

Because L_i has to select only one physical RAM as its final implementation, thus

$$I_i^{LUTRAM} + \sum_{j=1}^{N_B} I_i^{B_j} = 1 \tag{2}$$

We then need find the local optimal configuration for each physical RAM P_j to implement L_i. For simplicity, we can iterate all available width-depth configurations to find configurations that require the minimum number of P_j and minimum number of extra LUTs. We use RAM_i^j and $exLUT_i^j$ to denote the number of P_j and number of extra LUTs used in local optimal configuration for L_i with P_j. Thus the total number of extra LUTs used ($exLUT_i$) is

$$exLUT_i = exLUT_i^{LUTRAM} \cdot I_i^{LUTRAM} + \sum_{j=1}^{N_B} exLUT_i^{B_j} \cdot I_i^{B_j} \tag{3}$$

Then we can get the total number of extra LUTs used across the whole circuit

$$exLUT = \sum_{i=1}^{N_L} exLUT_i \tag{4}$$

Without considering the effect of any BRAM, we can get the total number of logic blocks (LB) used by this circuit

$$LB = LB_{logic} + \sum_{i=1}^{N_L} RAM_i^{LUTRAM} + \lceil exLUT/S_{LB} \rceil \tag{5}$$

[1]Using this simple scheme can already show good performance, as suggested by Prof. Vaughn Betz

Where LB_{logic} is the number LB used to implement circuit logic, S_{LB} is the size of logic block (which is 10 for Stratix-IV). The number of extra LUT is converted to the number LB with rounding up. To formulate the Eqn. 5 as the ILP, the rounding up operation $\lceil exLUT/S_{LB} \rceil$ can be replaced by another integer variable LB_{exLUT} with an additional constraint

$$LB = LB_{logic} + \sum_{i=1}^{N_L} RAM_i^{LUTRAM} + LB_{exLUT} \tag{6}$$

$$\text{s.t.} \qquad exLUT \leq S_{LB} \cdot LB_{exLUT} < exLUT + S_{LB} \tag{7}$$

We can also get the total number of each type of BRAMs used in the circuit in a similar way

$$BRAM_j = \sum_{i=1}^{N_L} RAM_i^{B_j} \tag{8}$$

When combining LB and all types of BRAMs together, we need to identify the dominating component that determines the number of LBs actually covered by the circuit (LB^*). Suppose there is one BRAM B_j for every K_j logic blocks. Then LB^* can be formulated as

$$LB^* = \max(LB, K_1 \cdot BRAM_1, K_2 \cdot BRAM_2, \ldots, K_{N_B} \cdot BRAM_{N_B}) \tag{9}$$

The Eqn. 5 can be reformulated as a series of equities for ILP.

$$\begin{cases} LB^* \geq LB, \\ LB^* \geq K_1 \cdot BRAM_1, \\ \qquad \vdots \\ LB^* \geq K_{N_B} \cdot BRAM_{N_B} \end{cases} \tag{10}$$

We can then formulate the objective function–the circuit area as

$$Area = A_{LB} \cdot LB^* + \sum_{j=1}^{N_B} A_{B_j} \cdot \left\lfloor \frac{LB^*}{K_{B_j}} \right\rfloor \tag{11}$$

Similar to Eqn. 6&7, we can replace $\left\lfloor \frac{LB^*}{K_{B_j}} \right\rfloor$ with $BRAM_j^*$ and additional constraints

$$\text{minimize} \qquad Aera = A_{LB} \cdot LB^* + \sum_{j=1}^{N_B} A_{B_j} \cdot BRAM_j^* \tag{12}$$

$$\text{s.t.} \quad LB^* - K_{B_j} < K_{B_j} \cdot BRAM_j^* \leq LB^*, \quad \forall j = 1, 2, \ldots, N_B \tag{13}$$

Finally, we convert the RAM mapping problem into a standard integer linear programming problem. Though ILP is a NP-hard problem, there are some well-studied heuristic algorithms and highly-optimized ILP solvers we can directly use. I choose Gurobi[2] as the ILP solver for this problem, because of Gurobi's high efficiency in solving ILP.

Algorithm complexity analysis. Since I choose a highly-optimized ILP solver for this problem, it is very difficult to make a precise analysis on the algorithm complexity. According to the official documents[3], Gurobi mainly uses a linear-programming based branch-and-bound algorithm to solve ILP. In the best case, ILP can be safely relaxed into LP problem and solved in $\mathcal{O}(n^{2.5})$ [1] (n is the number of decision variables). In the worst case, which is very rare and almost impossible, the algorithm has to traverse all nodes in the relaxation tree which can leads to the exponential complexity $\mathcal{O}(n^{2.5}2^n)$. However, it is still an open problem to make analysis on the amortized algorithm complexity for such ILP problem. Besides, multi-core parallelism in Gurobi can make such ILP solver run very efficiently.

Mapping Results. This RAM mapping is written in Python language. We that run the following command to generate RAM mapping file.

```
python ram_mapper.py --l_rams_path logical_rams.txt \
        --lb_cnt_path logic_block_count.txt \
        --phy_ram_cfg_path physical_rams.yaml \
        --map_out_path ram_mapping_1.txt
```

It take about 2.3 seconds to finish the RAM mapping for all 69 circuits, on UG machine's 4-core Intel i7-4790 CPU.

Table 1 shows the mapping results generated by the provided checker

```
./checker -d -t logical_rams.txt logic_block_count.txt ram_mapping_1.txt
```

The geometric average of the total FPGA area required over the benchmark set for this architecture is 2.003e+08, which can be treated as the lower bound of simple mapping scheme.

References

[1] Pravin M Vaidya. Speeding-up linear programming using fast matrix multiplication. In *30th annual symposium on foundations of computer science*, pages 332–337. IEEE Computer Society, 1989.

[2] https://www.gurobi.com/products/gurobi-optimizer/
[3] https://www.gurobi.com/resource/mip-basics/

A Mapping Results

Table 1: RAM mapping results for example Stratix-IV-like architecture (LB: Logic Block).

Circuit #	**LUTRAM Blocks Used**	**8K BRAMs Used**	**128K BRAMs Used**	**Regular LBs Used**	**Required LB Tiles in Chip**	**Total FPGA Area**
0	753	369	12	2941	3694	1.843e+08
1	1136	413	13	2988	4130	2.058e+08
2	0	62	0	1836	1836	9.162e+07
3	0	90	0	2808	2808	1.399e+08
4	154	806	26	7907	8061	4.022e+08
5	0	312	0	3692	3692	1.842e+08
6	0	185	6	1853	1853	9.245e+07
7	271	422	14	3947	4220	2.109e+08
8	92	543	18	5342	5434	2.715e+08
9	0	33	0	1636	1636	8.134e+07
10	560	203	6	1467	2030	1.008e+08
11	76	140	4	1329	1405	6.960e+07
12	0	43	0	1632	1632	8.119e+07
13	0	24	0	4491	4491	2.236e+08
14	125	195	6	1826	1951	9.709e+07
15	0	154	0	1956	1956	9.728e+07
16	0	88	0	2181	2181	1.087e+08
17	0	61	0	1165	1165	5.743e+07
18	0	156	6	2036	2036	1.010e+08
19	243	248	8	2236	2480	1.237e+08
20	16	268	9	2679	2700	1.349e+08
21	0	76	0	5100	5100	2.549e+08
22	536	296	9	2429	2965	1.474e+08
23	0	252	0	5230	5230	2.610e+08
24	30	435	14	4325	4355	2.172e+08
25	0	112	0	4517	4517	2.256e+08
26	458	218	7	1488	2180	1.087e+08
27	0	20	0	1496	1496	7.388e+07
28	301	236	7	2063	2364	1.173e+08
29	66	309	10	3025	3091	1.542e+08
30	0	215	0	5419	5419	2.707e+08
31	0	80	0	4347	4347	2.168e+08

32	814	454	15	3720	4540	2.268e+08
33	26	399	12	4006	4032	2.011e+08
34	51	416	14	1705	4200	2.099e+08
35	52	143	4	1376	1430	7.083e+07
36	861	446	18	1812	5400	2.699e+08
37	0	48	0	14969	14969	7.474e+08
38	0	281	9	3202	3202	1.594e+08
39	264	211	7	1845	2110	1.054e+08
40	0	179	0	3060	3060	1.528e+08
41	374	241	8	2035	2410	1.204e+08
42	0	90	0	1337	1337	6.638e+07
43	111	132	4	1212	1323	6.575e+07
44	0	205	7	2114	2114	1.056e+08
45	0	1	3	2782	2782	1.388e+08
46	564	398	13	3421	3985	1.989e+08
47	0	55	0	1439	1439	7.117e+07
48	0	574	20	6875	6875	3.428e+08
49	1235	1312	43	11883	13120	6.552e+08
50	0	580	0	11884	11884	5.935e+08
51	0	418	7	4204	4204	2.101e+08
52	0	641	0	9603	9603	4.800e+08
53	0	816	0	10817	10817	5.406e+08
54	0	885	0	10903	10903	5.447e+08
55	157	1049	34	10341	10498	5.238e+08
56	0	342	4	4578	4578	2.285e+08
57	0	466	0	7145	7145	3.564e+08
58	0	733	22	7700	7700	3.843e+08
59	4344	1798	59	13626	17980	8.980e+08
60	0	558	0	20371	20371	1.017e+09
61	0	1890	62	15079	18900	9.448e+08
62	43	493	16	4888	4931	2.461e+08
63	0	384	16	4846	4846	2.420e+08
64	843	1120	37	10451	11294	5.640e+08
65	0	357	0	12721	12721	6.355e+08
66	0	595	21	6310	6310	3.154e+08
67	1622	452	15	2894	4520	2.259e+08
68	0	192	0	4850	4850	2.423e+08
The Total CPU Runtime					**2.33 sec**	
Geometric Average of the Area					**2.00251e+08**	