Lichen Liu
1001721498

Assignment 3

ECE 1756H
November 30, 2022

1 Algorithm

1.1 Description

The RAM Mapper is a tool written in Python3 that takes the description for the logical circuits and hardware architecture as inputs and produces valid logical RAM to physical RAM mapping as output. The tool has a built-in FPGA area estimator for both providing heuristics during optimization and for final Quality of Result (QoR) reporting. The mapper has multiple stages of mapping and optimizations using different algorithms with different heuristics, targeting and optimizing specific aspects that may generate a mapping configuration that requires a lower FPGA area. Although the mapper operates in batch mode (i.e., can work on multiple logical circuits), circuits are individually and independently handled internally. The mapper is able to produce advanced mapping configuration features such as multilevel heterogeneous mapping configuration using different physical RAM types and shapes for a single logical RAM, and physical RAM sharing.

The mapper first generates an initial mapping configuration greedily. For each logical RAM, all possible configurations using a single type of physical RAM are generated. The candidates are then computed using the FPGA area estimator but assuming the entire circuit has only this logical ram. The candidate with the best area is picked. Other simpler area models were experimented with but none of them delivered the same optimal final QoR.

Next, the initial mapping is fed into a candidate-based simulated annealing engine. The engine swaps the current physical RAM configuration with other candidates and accepts the swap according to the delta in the area and current temperature. The general idea is that with a higher temperature in the early phases of optimization, a few bad swaps are deliberately accepted in the hope to jump out of the current local solution space. As optimization progresses, the temperature drops, hence gradually restricts the bad swaps accepted. Eventually, only good swaps are accepted when the temperature reaches zero. Moves are generated randomly, but with a 40% probability of picking a candidate with the physical RAM type not being the critical one in deciding the required LB count, and doing so benefits the QoR. As the multilevel configuration feature is supported, the engine can do the swap on any arbitrary end node (physical RAM configuration) in the configuration tree, by using the locator callback provided along with the physical RAM configuration candidate (similar to the ones used in the initial mapping phase). There are many details in the implementation worth mentioning. The engine can be configured to force into quenching mode (i.e., temperature set to zero) according to the current progress. The engine also dynamically extends the optimization steps if the current acceptance ratio is too high. Furthermore, there is a save-and-restore mechanism to make sure the best configuration ever reached is not thrown away. At the end of the simulated annealing flow, there is a quick fix-up convergence flow that keeps trying every candidate greedily until the circuit is converged to a local optimal solution. The heuristics used for deciding whether to accept the move is based on area. Rather than using the FPGA area, the required LB count is used as it is proportional to the FPGA area while

being faster in computing. However, there are still lots of moves that generate the same required LB count, so tie-breaking is simply done by counting the area of the extra LUTs and the RAM without considering the aspect ratio, which gives a noticeable boost in QoR.

There is another stage to split the existing physical RAM configuration into multiple physical RAM configurations in width dimension, as doing so won't inject any additional LUTs. The target here is to minimize the physical RAM width wasted by being the last one in the row (i.e., cliff). As long as the width of the original physical RAM configuration satisfies certain conditions, it is split. After the splitting, another round of simulated annealing is performed to actually fix the width wasted by the cliff.

Finally, physical RAM sharing is performed. It first looks for all physical RAM configurations using a single port, then finds eligible physical RAM that can serve as a provider. Next, all possible provider and receiver pairs are formed and are assigned with a heuristic delta cost which is the delta between LB and RAM areas without considering the aspect ratio. The solver picks the producer with the fewest number of receivers first and accepts the sharing of the candidate pair with the best delta cost.

1.2 Performance Optimization

The code is in Python and does millions of moves, therefore it is critical to optimize the performance of the code, otherwise running thousands of moves might already take an unacceptable amount of time. The code is profiled as frequently as adding new QoR optimizations using cProfile, and are optimized accordingly. As circuits are independent of each other, circuit-level parallelism is exploited by using the Pool from multiprocessing, which automatically detects the number of CPU cores to use. On the other hand, it has been identified that the FPGA area would be globally optimal when the number of tiles required equals the number of the regular logical blocks (not counting extra LUTs) from the user logical circuit input, therefore this condition is used for testing for early exit in several algorithms. Altogether, there is a huge performance gain from the techniques mentioned.

1.3 Computational Complexity

The computational complexity of candidate-based phases (i.e., initial mapping and simulated annealing) are all depending on the candidate sizes, which are both dependent on the number of logical RAMs and all possible physical shapes of the RAM architecture. The number of steps to perform for the simulated annealing engine is constant. Therefore the computational complexity here is $O(C \times R \times \sqrt{B})$, where C is the total number of circuits, R is the number of logical RAMs per circuit, and B is the number of bits of the physical RAM architecture.

The computational complexity of RAM splitting is directly linked with the number of logical RAMs, so having $O(C \times R)$.

RAM sharing, on the other hand, is forming pairs between all possible physical RAM config-

urations, therefore a complexity of $O(C \times R \times R)$. However, it is worth noting the quadratic complexity here does not contribute a lot to the total CPU runtime, and is heavily dependent on the mode of the logical RAM.

Summing all phases up, the computational complexity for the mapper program is: $O(C \times R \times (R + \sqrt{B}))$, where C is the total number of circuits, R is the number of logical RAMs per circuit, and B is the number of bits of the physical RAM architecture. Although RAM splitting is supported, it is only performed once so would not explode the number of physical RAM configurations created up.

2 Stratix IV QoR Results

Table 1: RAM mapping results for example Stratix-IV-like architecture

Circuit	LUTRAMs	8K BRAMs	128K BRAMs	Regular LBs	Required Tiles	FPGA Area
0	798	365	12	2946	3744	1.86717e+08
1	1512	464	15	3121	4640	2.31559e+08
2	0	62	0	1836	1836	9.16224e+07
3	0	81	1	2808	2808	1.3999e+08
4	201	800	27	7907	8108	4.05223e+08
5	0	276	4	3692	3692	1.84285e+08
6	3	183	4	1853	1856	9.25655e+07
7	659	464	15	4018	4677	2.33236e+08
8	130	547	17	5342	5472	2.73324e+08
9	0	33	0	1636	1636	8.13408e+07
10	564	205	6	1481	2050	1.01772e+08
11	108	142	2	1329	1437	7.10967e+07
12	0	11	2	1632	1632	8.11908e+07
13	0	24	0	4491	4491	2.23672e+08
14	270	207	6	1882	2152	1.07413e+08
15	0	90	4	1956	1956	9.7281e+07
16	0	56	2	2181	2181	1.0879e+08
17	0	61	0	1165	1165	5.74392e+07
18	0	156	6	2036	2036	1.01053e+08
19	360	260	8	2238	2600	1.29408e+08
20	113	221	7	2679	2792	1.39293e+08
21	0	60	1	5100	5100	2.54951e+08
22	609	306	9	2477	3086	1.53969e+08
23	0	107	11	5230	5230	2.61081e+08
24	56	439	14	4333	4390	2.18919e+08

25	0	112	0	4517	4517	2.25691e+08
26	458	219	7	1480	2190	1.09224e+08
27	0	20	0	1496	1496	7.38885e+07
28	529	276	9	2144	2760	1.37803e+08
29	106	311	10	3026	3132	1.56176e+08
30	0	215	0	5419	5419	2.70757e+08
31	0	80	0	4347	4347	2.16824e+08
32	1350	506	16	3913	5263	2.62609e+08
33	316	445	14	4143	4459	2.22086e+08
34	51	416	14	1705	4200	2.0996e+08
35	0	78	2	1360	1360	6.75333e+07
36	861	595	19	1788	5950	2.96734e+08
37	0	48	0	14969	14969	7.47457e+08
38	0	278	9	3202	3202	1.59477e+08
39	332	220	6	1871	2203	1.09808e+08
40	0	165	1	3060	3060	1.52801e+08
41	383	270	9	2030	2700	1.34974e+08
42	0	89	0	1337	1337	6.63812e+07
43	113	132	4	1212	1325	6.58346e+07
44	16	208	7	2114	2130	1.06395e+08
45	0	13	1	2782	2782	1.38822e+08
46	1172	476	14	3588	4760	2.37217e+08
47	0	54	0	1439	1439	7.11717e+07
48	3	689	22	6894	6897	3.43874e+08
49	1623	1350	45	11883	13506	6.75096e+08
50	0	580	0	11884	11884	5.93526e+08
51	0	419	8	4204	4204	2.1011e+08
52	0	641	0	9603	9603	4.80021e+08
53	0	816	0	10817	10817	5.40631e+08
54	0	884	1	10903	10903	5.44725e+08
55	178	1051	35	10341	10519	5.25709e+08
56	0	342	2	4578	4578	2.28558e+08
57	0	466	0	7145	7145	3.56439e+08
58	0	770	11	7700	7700	3.84359e+08
59	0	1402	38	11888	14020	7.00242e+08
60	0	558	0	20371	20371	1.01758e+09
61	0	1890	62	15079	18900	9.44819e+08
62	231	513	17	4902	5133	2.56478e+08
63	0	391	15	4846	4846	2.42065e+08
64	839	1123	37	10451	11290	5.63853e+08
65	0	357	0	12721	12721	6.35576e+08
66	101	641	21	6310	6411	3.20164e+08

67	1798	478	15	2966	4780	2.3816e+08
68	0	192	0	4850	4850	2.42312e+08

Geometric Average of the Total FPGA Area: 2.03794e+08

Command:

```
python3 -m ram_mapper --lb=logic_block_count.txt --lr=logical_rams.txt
↪  --out=mapping.txt
```

Total CPU runtime: 312.610 seconds

Environment: ug215 using 16 processes, 11th Gen Intel(R) Core(TM) i7-11700 @ 2.50GHz, 32GB RAM

3 Architecture Exploration

3.1 Without LUTRAM

BRAM Size	**Max Width**	**LBs / BRAM**	**Geometric Average FPGA Area (min width transistors)**
1 kbit	4	1	$2.4976e+08$
2 kbit	8	2	$2.27292e+08$
4 kbit	16	4	$2.17092e+08$
8 kbit	32	8	$2.17857e+08$
16 kbit	32	8	$2.19761e+08$
32 kbit	64	16	$2.40121e+08$
64 kbit	64	16	$2.75786e+08$
128 kbit	128	32	$3.44266e+08$

Table 2: Results without LUTRAM

From 2, the best average FPGA area of $2.17092e+08$ is achieved with the **4 kbit BRAM**, with a max width of 16 and LBs / BRAM ratio of 4. The optimal max width and LBs / BRAM ratio both increases as BRAM size increases. There is also a relative factor of 4 between the optimal max width and LBs / BRAM ratio. Max width is a key deciding factor of the area for BRAM, so for a smaller BRAM, a higher max width makes each bit more costly. Similarly for LBs / BRAM ratio, if the ratio is too large, the mapping solution might be bloating up the LB tiles required due to the aspect ratio constraint, therefore unnecessarily

wasting the area of unused regular LBs; whereas when the ratio is too small, circuits might be end up wasting the unused BRAM blocks. This ratio is all about the ratio between the number of regular LBs from the logical circuit and the number of logical memory bits. In other words, a larger physical RAM would typically be accompanied by more regular LBs.

3.2 With LUTRAM

BRAM Size	**Max Width**	**LBs / BRAM**	**Geometric Average FPGA Area (min width transistors)**
1 kbit	4	1	$2.40511e+08$
2 kbit	4	2	$2.15533e+08$
4 kbit	8	4	$2.04129e+08$
8 kbit	16	8	$1.99043e+08$
16 kbit	32	16	$2.013e+08$
32 kbit	64	32	$2.09588e+08$
64 kbit	64	32	$2.20513e+08$
128 kbit	128	64	$2.4039e+08$

Table 3: Results with LUTRAM

From 3, the best average FPGA area of $1.99043e+08$ is achieved with the **8 kbit BRAM**, with a max width of 16 and LBs / BRAM ratio of 8. Similar to the without LUTRAM case, the optimal max width and LBs / BRAM ratio both increases as BRAM size increases. Interestly, the ratio between are them are now 2. This implies fewer BRAMs are needed, which is due to the precense of LUTRAMs on the chip.

3.3 Customized Architecture

The customized architecture has the following checker string: **"-l 4 1 -b 8192 16 8 1"**, which translates to a LUTRAM with LBs / BRAM ratio of 4:1 and an 8 kbit BRAM with max width of 16 and ratio of 8:1. The geometric average area required is $1.94018e+08$. The architecture gives very efficient area because it has fewer LUTRAMs than the same architecture from 3, and the mapper is capable of recognizing this and hence using BRAMs instead thanks to the targeted physical RAM configuration moves in the simulated annealing engine. Requiring fewer LUTRAMs on the chip therefore reduces the LB areas even when the number of BRAMs stays the same.

4 Appendix

The source code can be found in my GitHub repo here: Source.