Spring 2021 ME/CS/ECE759 Final Project Report
University of Wisconsin-Madison

Multi-CPU and Multi-GPU Accelerated Logic Rewriting

Tianen Chen, Ruiqi Geng, Alliot Nagle

May 7, 2021

Abstract

We parallelized AIG-rewiring from the decades-old tool called ABC, developed from the Berkeley Logic Synthesis and Verification group. We aimed to implement and test the parallelization using both a multi-CPU approach with OpenMP and a multi-GPU approach with CUDA. We parallelized certain node cutting and AIG synthesizing functions, and found that OpenMP slightly improved the runtime, possibly due to limited parallelization and significant overhead. The CUDA approach proved particularly challenging, as the code base's interdependencies were far too complex to compile with CUDA integration within the time scope of this project. Further work is needed to unleash the potential for great speedup gains in both approaches using either parallelization method.

Link to Final Project git repo: https://euler.wacc.wisc.edu/rgeng5/me759-rgeng5.git

Contents

1. General information

In this short section, please provide only the following information, in bulleted form (four bullets) and in this order:

1. Your home department: ECE, Medical Physics, ECE
2. Current status: PhD student
3. Individuals working on the Final Project (include yourself)
    - Tianen Chen
    - Ruiqi Geng
    - Alliot Nagle
4. Choose one of the following two statements (there should be only one statement here):
    - I am not interested in releasing my code as open source code.

2. Problem statement

An AND-Inverter Graph (AIG) is a directed, acyclic graph that represents a structural implementation of the logical functionality of a circuit or network [1]. Fig. 1 shows the AIG of an arbitrary circuit with six primary inputs (a to f) and one primary output (F). This AIG has 13 nodes that are labeled from 8 to 20. In an AIG, nodes are 2-input AND gates, and dashed lines indicate inverted edges.

Logic rewriting is a greedy technique to optimize a given circuit for logic synthesis [2]. Logic rewriting optimization functions can operate on various graph types including AIGs. The standard logic rewriting operation preserves the logic level and aims to reduce the number of nodes of a subgraph corresponding to the cut under analysis. Fig. 2 shows the moment that node 13 is being visited. A 4-input cut of this node is shown by a green color in this figure. The function of node 13 based on inputs of this cut is: !(!n8*!n9)*(e*n11). The corresponding subgraph for this cut is shown in Fig. 2, which consists of nodes 10, 12, and 13.

An open-source logic synthesis and verification tool, based on CPU, called ABC can achieve standard rewriting function on AIGs. A full episode of ABC rewrite consists of (i) compute 4-feasible cuts for all nodes; (ii) visit each node and obtain its Boolean function based on inputs of its cuts; (iii) look up the Boolean function in a precomputed database of 4-input functions; (iv) try different rewritten versions of this function in the database, as (v) an AIG subgraph for each rewritten version of the Boolean function is generated inside the rewrite function. (vi) If the node count of this new subgraph is fewer than the original one, the rewrite operation is accepted, and the subgraph is substituted.

The main goal of this contest problem is to implement a rewriting optimization function using both OpenMP and CUDA in order to increase its speed through multi-CPU and multi-GPU implementations.

Motivation/Rationale:

Developing a logic rewriting function in CUDA is an open problem which has been outsourced in the CAD competition for the 2021 International Conference on Computer Aided Design (ICCAD) [3]. We are motivated to find a solution to this open problem with the goal of submitting our solution if we are successful in achieving the state-of-the-art results or better from the current CPU implementation. More information about the contest is available here: http://iccad-contest.org/2021/

Accelerating logic rewriting using multi-CPU and multi-GPU approaches also relates to some of the research interests of the group. In particular, logic rewriting has relevance to designing logic-based neural networks for low-power inferencing applications. Once a neural network has been fully trained, it can be used for inferencing on new data. In certain applications, it may be necessary to perform this inferencing in a low-power setting such as on a mobile device. Every neuron in the neural network can be converted into a real-valued logic function which of course can be optimized to be rewritten in a smaller logic function. This approach enables lower memory usage and higher energy efficiency of the network.

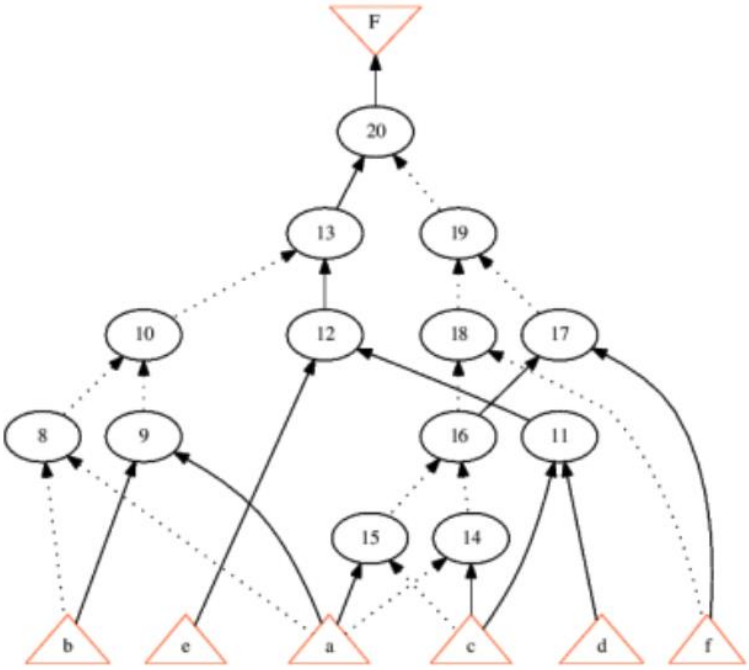

Fig. 1: An example AIG with 13 nodes for the following 6-input 1-output Boolean expression: F = (!(!(!a*!b)*!(a*b))*(e*(c*d)))*!(!(!(!(a*!c)*!(!a*c))*!f)*!((!(a*!c)*!(!a*c))*f)), where ! means NOT or inversion and * means AND.

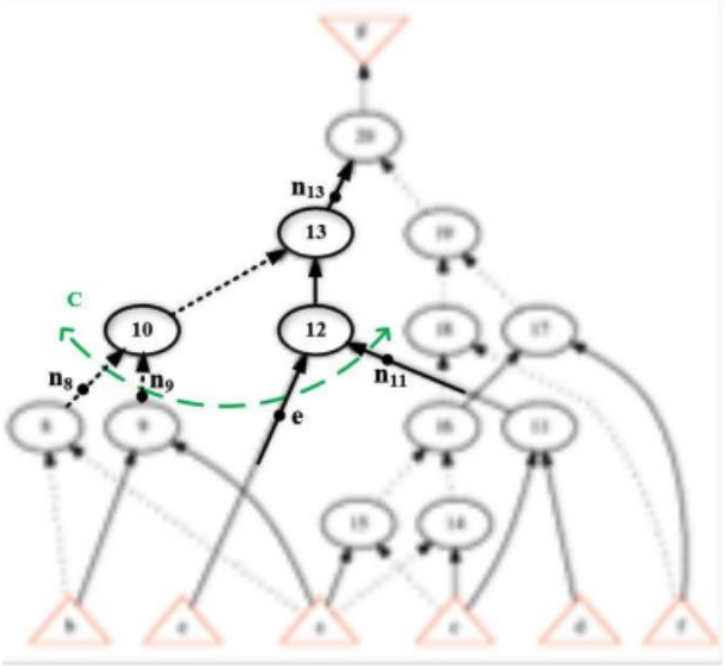

Fig. 2: A 4-input cut (4-cut) of node 13 (n_{13}) and its current Boolean function based on inputs of this cut: $n_{13} = !(!n_8 * !n_9) * (e * n_{11})$. The corresponding sub-graph of this cut is also shown.

3. Solution description

Our proposed method was to implement a naive distributed lookup table similar to the tool ABC. We primarily used CUDA and OpenMP. In ABC, each node and 4-input cut is a boolean function that is checked for in a look-up table. The boolean function is part of a subgraph of the AIG, and if the boolean function can be replaced such that the subgraph contains fewer nodes, then the subgraph is permanently rewritten.

For our final project, we considered OpenMP and CUDA as two different approaches to parallelizing the AIG rewriting in general, and the look-up table in particular. Each approach provides a different level of granularity for satisfying the goal of parallelizing the AIG rewrite step: OpenMP threads are better suited for handling more complex tasks, so we mostly delegate OpenMP threads to graph traversal. On the other hand, CUDA is more appropriate for finer levels of granularity, particularly for tasks such as the truth table look-ups in src/misc/extra/extra.h.

The source code for rewriting an AIG is scattered throughout the src/ directory, but we focused our efforts on select source code files that have the most impact on performance for purposes of parallelization: these files are src/base/abci/abcQuant.c, src/opt/cut/cutNode.c, and src/misc/extra/extra.h. Abc_NtkRewrite() (located on line 62 of src/base/abci/abcRewrite.c) is the starting point for rewriting and AIG.

General structure of the code involves: (i) reading in the input AIG graph, (ii) node rewriting, and (iii) network verification to make sure the abbreviated AIG is identical to the input AIG [4,5].

OpenMP implementation:

We parallelized these functions because they were taking a large portion of time according to profiling:

- Abc_NtkSynthesize(): This function is the starting point for synthesizing the AIG, and it therefore calls Abc_NtkRewrite() to convert the user-provided AIG into an AIG with a reduced number of and nodes. In the function Abc_NtkTransRel (see line 160 of src/base/abci/abcQuant.c) we parallelize the for loops on lines 216 and 234. The first for loop calls Abc_NtkSynthesize(), which means that the for loop performs a rewriting of the entire network for every quantification of each primary input. The second (following) for loop performs a verification step to ensure that each of the primary inputs is connected to the rest of the graph (i.e., each primary input has a non-zero fan out).
- Cut_NodeDoComputeCuts(): This function can be found by following the function calls made by the Abc_NtkSynthesize() function described above. Beginning at line 621 of src/opt/cut/cutNode.c, we added four OpenMP sections for each consideration of merging two small cuts, a large cut and a small cut, and two large cuts. One of these four merge candidates is accepted before returning and continuing with the graph rewriting. Since we dealt with linked list data, we used OpenMP *task* and *private* to avoid false sharing and race conditions.
- Within src/misc/extra/extra.h there are many functions for the purposes of truth table lookup, and these functions are important for providing speedup for rewriting [1]. We parallelized a handful of these functions, namely Extra_TruthCopy(), Extra_TruthNot(), Extra_TruthAnd(),

Extra_TruthOr(), and Extra_TruthNand(), which are all called in Cut_TruthCompute(), which is called by Cut_NodeDoComputeCuts() (the function described in the bullet point above). For example, for one of our loops, we used:

```
#pragma omp parallel for simd reduction (+:Counter)
        for ( w = Extra_TruthWordNum(nVars)-1; w >= 0; w-- )
                Counter += Extra_WordCountOnes(pIn[w]);
        return Counter;
```

and the -ftree-vectorize flag in compilation to enable *simd*.

We used *gprof* to profile the OpenMP implementation [6]. *Gprof* is a profiling program which collects and arranges statistics on your programs. It looks into each of our functions and inserts code at the head and tail of each one to collect timing information. Then, when we run our program normally, it creates "gmon.out" -- raw data which the *gprof* program turns into profiling statistics.

CUDA implementation:

Our overarching goal with the CUDA implementation was to parallelize the “rewrite” function for AIGs built into ABC. From digging through dependencies, a majority of the parallelization opportunity lies within src/misc/extra/extra.h. Within extra.h there are many comparison functions that perform simple for loops and operate on two sets of arrays. From there, we wanted to implement the kernel.

4. Overview of results. Demonstration of your project

OpenMP implementation:

Reading, rewriting, and network verification:

```
UC Berkeley, ABC 1.01 (compiled May  7 2021 06:28:53)
abc 01> read_aiger i10.aig
abc 02> print_stats
i10                          : i/o =  257/  224  lat =    0  and =   2675  lev = 50
abc 02> rewrite
abc 02> write_aiger i10_rewrite.aig
abc 02> print_stats
i10                          : i/o =  257/  224  lat =    0  and =   2096  lev = 48
abc 02> ***EOF***
i10                          : i/o =  257/  224  lat =    0  and =   2396  lev = 37
i10                          : i/o =  257/  224  lat =    0  and =   1851  lev = 35
Networks are equivalent.  Time =     0.29 sec
Reading =  10.92 milli sec   Rewriting = 162.96 milli sec   Verification = 326.81 milli sec   Total = 500.70 milli sec
```

Fig. 3. output of *vim OMPout*.

Input and output nodes and their connections:

```
# Benchmark "i10" written by ABC on Thu May  6 21:18:47 2021
.model i10
.inputs V32(0) V32(1) V32(2) V32(3) V56(0) V289(0) V10(0) V13(0) V35(0) \
 V203(0) V288(6) V288(7) V248(0) V249(0) V62(0) V59(0) V174(0) V215(0) \
 V66(0) V70(0) V43(0) V214(0) V37(0) V271(0) V40(0) V45(0) V149(7) V149(6) \
 V149(5) V149(4) V1(0) V7(0) V34(0) V243(0) V244(0) V245(0) V246(0) V247(0) \
 V293(0) V302(0) V270(0) V269(0) V274(0) V202(0) V275(0) V257(7) V257(5) \
 V257(3) V257(1) V257(2) V257(4) V257(6) V9(0) V149(0) V149(1) V149(2) \
 V149(3) V169(1) V165(0) V165(2) V165(4) V165(5) V165(6) V165(7) V165(1) \
 V88(2) V88(3) V55(0) V169(0) V52(0) V5(0) V6(0) V12(0) V11(0) V4(0) \
 V165(3) V51(0) V65(0) V290(0) V279(0) V280(0) V288(4) V288(2) V288(0) \
 V258(0) V229(5) V229(4) V229(3) V229(2) V229(1) V229(0) V223(5) V223(4) \
 V223(3) V223(2) V223(1) V223(0) V189(5) V189(4) V189(3) V189(2) V189(1) \
 V189(0) V183(5) V183(4) V183(3) V183(2) V183(1) V183(0) V239(4) V239(3) \
 V239(2) V239(1) V239(0) V234(4) V234(3) V234(2) V234(1) V234(0) V199(4) \
 V199(3) V199(2) V199(1) V199(0) V194(4) V194(3) V194(2) V194(1) V194(0) \
 V257(0) V32(8) V32(7) V32(6) V32(5) V32(4) V32(11) V32(10) V32(9) V88(1) \
 V88(0) V84(5) V84(4) V84(3) V84(2) V84(1) V84(0) V78(5) V78(4) V2(0) V3(0) \
 V14(0) V213(0) V213(5) V213(4) V213(3) V213(2) V213(1) V268(5) V268(3) \
 V268(1) V268(2) V268(4) V8(0) V60(0) V53(0) V57(0) V109(0) V277(0) V278(0) \
 V259(0) V260(0) V67(0) V68(0) V69(0) V216(0) V175(0) V177(0) V172(0) \
 V171(0) V50(0) V63(0) V71(0) V292(0) V291(0) V91(0) V91(1) V294(0) V207(0) \
 V295(0) V204(0) V205(0) V261(0) V262(0) V100(0) V100(5) V100(4) V100(3) \
 V100(2) V100(1) V240(0) V242(0) V241(0) V33(0) V16(0) V15(0) V101(0) \
 V268(0) V288(1) V288(3) V288(5) V301(0) V108(0) V108(1) V108(2) V108(3) \
 V108(4) V108(5) V124(5) V124(4) V124(3) V124(2) V124(1) V124(0) V132(7) \
 V132(6) V132(5) V132(4) V132(3) V132(2) V132(1) V132(0) V118(5) V118(4) \
 V118(3) V118(2) V118(1) V118(0) V118(7) V118(6) V46(0) V48(0) V102(0) \
 V110(0) V134(1) V134(0) V272(0) V78(2) V78(3) V39(0) V38(0) V42(0) V44(0) \
 V41(0) V78(1) V78(0) V94(0) V94(1)
.outputs V321(2) V356 V357 V373 V375(0) V377 V393(0) V398(0) V410(0) \
 V423(0) V432 V435(0) V500(0) V508(0) V511(0) V512 V527 V537 V538 V539 V540 \
 V541 V542 V543 V544 V545 V546 V547 V548 V572(9) V572(8) V572(7) V572(6) \
 V572(5) V572(4) V572(3) V572(2) V572(1) V572(0) V585(0) V587 V591(0) \
 V597(0) V603(0) V609(0) V620 V621 V630 V634(0) V640(0) V657 V707 V763 V775 \
 V778 V779 V780 V781 V782 V783 V784 V787 V789 V798(0) V801 V802(0) V821(0) \
 V826(0) V966 V986 V1213(11) V1213(10) V1213(9) V1213(8) V1213(7) V1213(6) \
 V1213(5) V1213(4) V1213(3) V1213(2) V1213(1) V1213(0) V1243(9) V1243(8) \
 V1243(7) V1243(6) V1243(5) V1243(4) V1243(3) V1243(2) V1243(1) V1243(0) \
 V1256 V1257 V1258 V1259 V1260 V1261 V1262 V1263 V1264 V1265 V1266 V1267 \
 V1274(0) V1281(0) V1297(4) V1297(3) V1297(2) V1297(1) V1297(0) V1365 V1375 \
 V1378 V1380 V1382 V1384 V1386 V1387 V1392(0) V1423 V1426 V1428 V1429 V1431 \
 V1432 V1439(0) V1440(0) V1451(0) V1459(0) V1467(0) V1470 V1480(0) V1481(0) \
 V1492(0) V1495(0) V1512(3) V1512(2) V1512(1) V1536(0) V1537 V1539 V1552(1) \
 V1552(0) V1613(0) V1613(1) V1620(0) V1629(0) V1645(0) V1652(0) V1669 \
 V1671(0) V1679(0) V1693(0) V1709(4) V1709(3) V1709(2) V1709(1) V1709(0) \
 V1717(0) V1719 V1726(0) V1736 V1741(0) V1745(0) V1757(0) V1758(0) V1759(0) \
 V1760(0) V1771(1) V1771(0) V1781(1) V1781(0) V1829(9) V1829(8) V1829(7) \
 V1829(6) V1829(5) V1829(4) V1829(3) V1829(2) V1829(1) V1829(0) V1832 \
 V1833(0) V1863(0) V1864(0) V1896(0) V1897(0) V1898(0) V1899(0) V1900(0) \
 V1901(0) V1921(5) V1921(4) V1921(3) V1921(2) V1921(1) V1921(0) V1953(1) \
 V1953(7) V1953(6) V1953(5) V1953(4) V1953(3) V1953(2) V1953(0) V1960(1) \
 V1960(0) V1968(0) V1992(1) V1992(0) V650 V651 V652 V653 V654 V655 V656 \
 V1370 V1371 V1372 V1373 V1374
```

Fig.4. output of *vim results.blif.*

Timing for different multi-CPU parallelization settings:

		Execution time (milli seconds)			
	# of nodes	reading	rewriting	network verification	total time
no parallelization	1	10.4	166	335	512
pthreads	1	10.8	162	328	501
OpenMP	1	10.4	162	333	506
	2	9.2	162	330	502
	4	10.9	163	327	501
	8	10.7	162	329	501
	16	10.4	162	329	502

Compared to no parallelization, turning on pthreads improved the runtime by ~11 ms. OpenMP also improved the runtime, by 6-11 ms, with similar performance to pthreads. OpenMP did not accelerate the execution more with more threads activated, because the parts we parallelized did not take a lot of time and the overhead of activating the threads outweighed the benefits of parallelization. As shown in the profiling results, many other functions that we didn't get to parallelize took a major part of the time.

```
:ach sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 37.50      0.03     0.03        2    15.00    15.00  Gia_ManSimTry
 25.00      0.05     0.02   339458     0.00     0.00  Abc_NtkSynthesize
 12.50      0.06     0.01    51497     0.00     0.00  Saig_ManBmcCreateCnf_rec
 12.50      0.07     0.01    24445     0.00     0.00  Io_ReadBlifNetworkConnectBoxesOneBox
 12.50      0.08     0.01                             Io_BlifHashLookup
  0.00      0.08     0.00   145185     0.00     0.00  Wlc_PrsWriteErrorMessage
  0.00      0.08     0.00   115992     0.00     0.00  Wlc_ReadSmtBuffer
  0.00      0.08     0.00    95685     0.00     0.00  Gia_ManCorrSpecReduceInit
  0.00      0.08     0.00    68679     0.00     0.00  Acb_ObjRemoveConst
```

Fig.5. Profiling result.

CUDA trial and error:

Our progress was immediately halted by the complexity of the compilation of a code base of this magnitude (1759 source files). Our problem and troubleshooting progress is documented below:

1. Compiling with CUDA was extremely difficult for a code base of this complexity. Namely, we were trying to restructure the code base and makefile by changing all dependencies from ".cpp", ".c", and ".h" to ".cu" and ".cuh".
    a. This proved to be especially unproductive since there are many interdependencies within the code that cannot be compiled without massive overhaul. The makefile itself is dense, and we struggled with compilation throughout the production of this code.
2. There were numerous errors within the makefile that we struggled to debug.
    a. The flags and output targets for nvcc compiler were difficult to troubleshoot.
    b. We were unfamiliar with the correct types of linker flags when combining C, C++, and CUDA files.
3. The biggest challenge to compiling the reworked CUDA code ABC was the addition of the dependencies within every single file.
    a. For example, every .c file now contained through the "includes" statement a reference to a potential CUDA file of ".cuh" extension.
    b. Our first fix for this problem was the substitution of files that relied on dependency to extra.h. This approach failed due to the numerous interdependencies that lied within the code base. In short, the entire project includes one file or another from a different folder that is directly touched by extra.h. Therefore, it becomes difficult to compile.
    c. Our second fix was the renaming and restructuring of all #include directives to ".cu" and ".cuh" files. This proved to be effective up to the point where errors began appearing within the gcc compiler code. From there, we halted progress on compilation and began to shift toward restructuring the code.
4. We also tried the CMake, and with the BLT package [7, 8] to no avail. BLT is a composition of CMake macros and several widely used open source tools assembled to simplify HPC software development.

CUDA final error message:

```
In file included from /usr/include/sys/stat.h:446,
                 from src/misc/util/utilFile.cu:26:
/usr/local/gcc/x86_64/10.2.0/lib/gcc/x86_64-pc-linux-gnu/10.2.0/include-fixed/bits/statx.h:38:25: error: missing binary operator before token
"("
   38 | #if __glibc_has_include ("__linux__/stat.h")
      |                         ^
```

Fig.6. CUDA compilation error.

5. Deliverables

OpenMP implementation:

From git repo, pull me759-rgeng5/FinalProject/openMP/abc/ to Euler;

cd to the directory called abc/, where Makefile and run_abc.sh are located;

sbatch run_abc.sh (make -j 16 accelerated the make process by using 16 compile jobs);

vim OMPout (shows the timing results);

Compilation and running may take 10-15 minutes.

Inputs:

source files in .c, .cpp, and .h are in src/ ;

i10.aig--the test AIG graph provided.

Outputs:

OMPabc_compiled.out, OMPabc_compiled.err--compilation outputs;

OMPout--number of nodes after rewrite, network verification, and timing for each mode of parallelization (Fig.3);

results.blif--shows the input and output nodes and their connections (Fig.4);

OpenMP profiling:

gmon.out

gprofreport_no--no parallelization;

gprofreport_PT--with pthreads;

gprofreport_OMP_${nodes}--with OpenMP using ${nodes} number of threads (Fig.5)

6. Conclusions and Future Work

Summarize the lessons learned and the highlights of your project work. Explain what remains to be done in the future, and how hard it would be to accomplish what is left at this point.

We successfully implemented and tested a preliminary multi-CPU approach with OpenMP to accelerate logic rewriting. Throughout the process, we came to a deeper appreciation of the amount of thoughts and effort behind common industry tools like ABC. Along the way, this project has left us with some valuable lessons:

1. CUDA compilation is a tricky process and makefiles can be hard to rearrange if the dependencies are not accounted for.
2. The ABC tool has many parallelizable sections. Simple for loops that perform repetitive actions can and should be easily parallelized.
3. Extremely large and complex code bases can be broken down to simpler sections in order to see opportunities for improvement.

Moving forward, we would like to try the following approaches:

1. CUDA compilation should be restructured around extra.h.
    a. Instead of restructuring the entire code base, we want to build a wrapper that calls an external .cu function. Therefore, we don't need to recompile the entire code base with nvcc and can instead just recompile the extra.h and its external CUDA function that contains the kernel.
    b. Try CMake with the BLT package [7, 8]
2. Change the algorithm to reply less on linked list data to utilize more threads to accelerate further. That would take more analysis of the complex code structure
3. Parallelize the verification part of the code with OpenMP, since that's taking a major part of the execution time
4. Test the correctness of the resultant program, which would be easy to do with test AIG.

References

[1] Berkeley Logic Synthesis and Verification Group, ABC: A System for Sequential Synthesis and Verification, Release YMMDD. http://www.eecs.berkeley.edu/~alanmi/abc/

[2] Wikipedia. And-inverter graph. https://en.wikipedia.org/wiki/And-inverter_graph.

[3] Ghasem Pasandi, Sreedhar Pratty, David Brown. Problem C: GPU Accelerated Logic Rewriting. ICCAD-2021 contest.

[4] ABC documentation: http://www.ece.ubc.ca/~eddieh/html/abc_dox/main.html

[5] ABC functions: http://eddiehung.github.io/dox-abc/dd/d04/abcQuant_8c.html#acf6222569c564781c16cc4f31e59c605

[6] David August. Make and Gprof. https://www.cs.princeton.edu/courses/archive/fall07/cos217/lectures/16MakeGprof.pdf

[7] Building Cross-Platform CUDA Applications with CMake https://developer.nvidia.com/blog/building-cuda-applications-cmake/

[8 BLT: Build, Link, and Test https://llnl-blt.readthedocs.io/en/develop/index.html