VLSI Circuit Partitioning

Implementation using Kernighan-Lin Algorithm in C++

Rafal Piersiak
Stony Brook University

Abstract—**Partitioning in automated VLSI design is utilized to minimize the interconnects between partitions, area, propagation delay, terminals, and the number of partitions in order to design a circuit that can meet high performance requirements. This paper will present the implementation of the Kernighan-Lin algorithm in C++, discuss the performance of the code, and the resulting output data.**

Keywords— VLSI; Kernighan-Lin; Partitioning; K-L algorithm; Bisectioning; Interconnects;

I. Introduction

VLSI designs of high complexity can be efficiently managed with the institution of partitioning, which decomposes any complex system into more manageable pieces, or partitions [4]. Partitioning assures that the system will maintain its original functionality, minimize interconnects between partitions, and carry a small weight in terms of total design time [4].

The theory of partitioning is effectively used in assemblies, such as computers for instance. By building an assembly out of smaller components, the ease of design increases due to parallelization, where the subsystems of the assembly can be designed and manufactured separately [4]. The same holds true for VLSI designs, where multiple chips may be necessary if the design is too large, or the design can be partitioned internally to speed up the design cycle.

There are three levels of partitioning. System level partitioning defines a set area and terminal count for a particular design. The design may not leave the prescribed bounds. Board level partitioning focuses primarily on board space and begins to factor in propagation delays between chips. Because chips can vary in size, terminal counts are not stressed. Chip level partitioning reemphasizes terminals, by ensuring that connections between partitions does not exceed the terminal count [4]. At this level, propagation delay can be a major performance damper, depending on how the critical path is assessed. By placing components with large propagation delays in the same partition, the threat of degraded system performance can be minimized [4]. Although area is sometimes considered in chip level partitioning, it is mostly a novelty, compared to minimalizing partition interconnects and critical paths.

VLSI design primarily employs chip level partitioning to minimize the number of interconnects between partitions. A VLSI design can be considered as a collection of vertices (the inputs/outputs of a component), with each vertex having a set of edges (wire between inputs/outputs of the components) connected to it, in the format G = (V, E). By reducing the number of interconnects between partitions, the amount of area utilized by the interconnects is minimized, resulting in a smaller design. Smaller designs naturally present shorter delays, as well as more efficient fabrication and independent design [4].

The implementation of this concept can be described in mathematical form, where the number of interconnects between partitions is called the cut size. Minimization of the cut size is called the mincut problem. Each edge has a cost, c_{ij}, where i and j describe the nodes between the edge. [1] The function used to find the mincut is presented in [4]:

$$\sum_{i=1}^{k}\sum_{j=1}^{k} c_{ij}\,,(i \neq j) \qquad \text{is minimized} \qquad (1)$$

II. Related Work

There are two prominent algorithms that work to solve the mincut problem. They are the Kernighan-Lin and the Fiduccia-Mattheyses algorithms, where the Fiduccia-Mattheyses algorithm is a modification of the earlier Kernighan-Lin algorithm. Both will be presented in this section.

A. The Kernighan-Lin Algorithm

B. W. Kernighan and S. Lin introduced their partitioning algorithm in a paper, titled “An Efficient Heuristic Procedure for Partitioning Graphs.” The Kernighan-Lin Algorithm was first published in 1970, in the Bell System Technical Journal. The Kernighan-Lin algorithm is a bisectioning algorithm, which creates subsets of equal size [4]. The algorithm swaps a pair of vertices, one from each partition, and looks for an improvement in the cut size. If an improvement, is found the algorithm computes the new gain values, and proceeds to swap another pair of vertices. This repeats until the cut size increases, meaning that the previous swap solved the mincut problem. A simple representation of the algorithm is available in Figure 1.

The gain values, D(i) for both partitions is calculated using the formula:

$$D(i) = inedge(i) - outedge(i) \qquad [2]$$

where the inedge represents the number if edges going into the vertex, and outedge represents the number of edges going out of the vertex. This is done for all vertices in both partitions for each iteration of the Kernighan-Lin algorithm.

The gain values, G(i) is calculated using the formula:

$$G(i,j) = D(i) + D(j) - 2c_{ij} \qquad [3]$$

where D(i) and D(j) were calculated in the previous step. The cost, c_{ij} is the number of edges connecting to node i and j that cross the imaginary line between the two partitions.

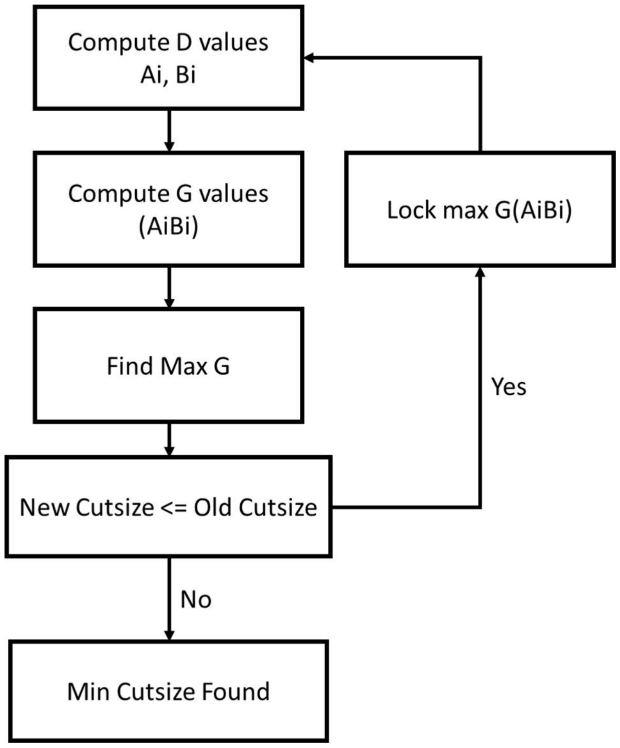

Figure 1: Kernighan-Lin Flowchart

The new cutsize is calculated by taking the previous cutsize and subtracting the max gain, G, found in the following formula:

$$New\ Cutsize = Previous\ Cutsize - G(i,j) \qquad [4]$$

If the new cutsize is at least as small as the previous cut size or smaller, the two vertices that were just swapped, are "locked," meaning they will not be swapped anymore because they are placed optimally in their respective partition. The algorithm will repeat until the new cut size is larger than the previous cut size, where G(i,j) is found to be a negative value. This shows that there is no further improvement and the previous iteration of the K-L algorithm has provided the optimal solution to the partitioning problem.

The K-L algorithm is robust, but suffers from several severe pitfalls that disqualify it from use in even moderate sized circuits [4]. The authors note that the time complexity of the algorithm is on the order of $O(n^3)$, which is the primary reason this algorithm is not relevant today [1]. Other reasons include, the lack of ability to specify hypergraphs, where an edge is specified to be connected to a set of vertices [4]. The K-L algorithm doesn't allow for unequal partitions, nor does it allow arbitrarily weighted graphs, meaning to define costs for edges [4].

The user can specify whether certain vertices need to stay together before the initial run of the K-L algorithm, as long as they represent a small portion of the total partition. This is usually done to keep certain circuits together, rather than splitting up their components across the two partitions [4].

B. *The Fiduccia-Mattheyses Algorithm*

C.M. Fiduccia and R.M. Mattheyses introduced their partitioning algorithm in a paper, titled "A Linear-Time Heuristic for Improving Network Partitions." The Fiduccia-Mattheyses Algorithm was first published in 1982, and presented at the 19th Design Automation Conference.

The Fiduccia-Mattheyses (F-M) algorithm is an improvement on the K-L algorithm described twelve years before. It carries out a number of new improvements, which primarily speeds up the execution of the algorithm. The F-M algorithm allows for one vertex to be moved across between partitions [4]. This allows for faster function because vertices don't need to be traded in order to facilitate a smaller cut size. Another feature, is the implementation of the concept of cut size to hypergraphs, which facilitates the transfer of one vertex to another partition [4]. Two features not available in K-L, unbalanced partitions and vertex weights are fully implemented in the F-M algorithm, which allows for more flexibility in how the vertices are partitioned.

The primary data structure implemented in the F-M algorithm uses a bucket list. The list contains sets of vertices sorted based on their vertex gains. During each move it is expected that the cut size decrease, but sometimes the gain may be negative. Such moves are allowed to pass in order to facilitate a "climb out of a local minima" [3]. It is also important to monitor the balance between the partitions. Transferring too many nodes to one partition could cause run-away loops as well defeat the purpose of the algorithm, because the lowest cut size is to have all vertices located in one partition. Moves continue until there is no improvement in cut size (all cells become locked) or partition balance is limits further movement [3].

Performance benefits occur due to the minimized recalculation of gains. Only gains of those neighboring a moved cell need to be recomputed. These gain recalculations are implemented in such a way, that only simple increments/decrements are required.

Although the relative speed of the F-M algorithm is a huge improvement compared to the K-L algorithm, its initial programming complexity can lead to the decimation of these speed improvements, if great care is not taken. One such situation is to ensure that only free neighbors have their gains updated, which when implemented carefully will avoid the $O(p^2)$ work [3]. Other situations are described as well and are handled automatically in the F-M algorithm by the implementation of the bucket list, which reduces the overall complexity of the formulation of the algorithm.

III. Proposed Solution

The implementation the K-L algorithm described in this paper is written in C++ using the C++11 coding standard. The program consists of a main file and five secondary files that contain the functions and variables called out in the main program. Each secondary file has its own header file, bringing the total file count for this program to eleven files. This was done to aid readability and provide a formal structure to the program.

The initial iteration of the program can be described in the following manner:

1. Parse input .netD file
2. Randomize two equal partitions of vertices
3. Calculate the initial cut size
4. Calculate the D values
5. Calculate the initial G values
6. Find the max G
7. Calculate the new cutsize
8. Check if the new cutsize is an improvement

After the initial iteration of the program, steps 4-8 are repeated until the new cut size shows sign of no improvement, meaning it has increased compared to the previous cut size.

The parse function takes in the input .netD file and creates a 2D vector with N rows, where N is the total amount of cells in the file. As the file is read, each source node has its associated sink connections inserted into the respective row of the 2D vector. The size of each row is equal to the degree of the vertex. During parsing, the .netD file only specifies an edge in one direction [2]. For example, if vertex 0 is connected to vertex 4, an entry will be pushed into row 0, with the value of 4. But this only represents half of the information. To correct for this misrepresentation, an entry is placed in row 4, with a value of 0. This ensures that for any vertex, we know each other vertex it is connected to.

The randomize function creates a sorted vector with entries from 0 to N – 1. The shuffle function is called to randomize the entries in this vector. Half of this vector is copied into a vector designated as partition A, and the other half is copied into a vector designated as partition B.

The generateG1 function creates an N x N + 4 array, which contains the D values for all he vertices. Using an N x N array facilitates the ease of calling a vertex[i][j]. The addition of the extra 4 columns is used to store whether the vertex is in partition A or B, the amount of inside and outside edges, and the D gain of the vertex. The 2D vector contains one of four values that signify, whether the source vertex (row) is in A or B and whether its sink vertex (column) is in A or B. This value is used to determine how to compute the D gain for that source vertex. During initialization, all vertices have their D gain values updated. Once the next iteration takes place, only the D gain values of the swapped vertices is altered, and all respective sink vertices associated with them. This eliminates a major chunk of time that could be wasted. The function exits and returns the initial cut size during the initial run. The returned value is not used for subsequent runs.

The generateG2 function calculates the G gain values for all the possible combinations, N^2. If there are four vertices in each partition, the G gain will be calculated 16 times. The function reads the value at vertex[i][j] along with the D gain of vertex I and vertex j and calculates the G gain. The function exits and returns a struct containing the max G and the two vertices responsible for the gain.

An output file is generated that prints out the initial partition and the final partition with minimal cut size. Total execution time is also printed in the file as well.

A. Relevant Implementation Issues of Proposed Solution

The parsing speed of the .netD file is almost instantaneous. All vertices are assigned their respective connections to other vertices quickly with minimal storage. The parsing function can be coded to remove the creation of a 1D vector that stores the number of connections each vertex contains, because the size of the row of the 2D vector can be utilized for that purpose.

Generation of the D gain values for all connections in the N x N + 4 vector created in the generateG1 function takes a few seconds for the smallest benchmark file. The same goes for the generation of the G gain values in the generateG2 function. Because the vertices are tied to the indexes of the 2D vectors, there is no need to search through the row to find a particular connected vertex, which increases the speed of the program.

Although this seems like a very convenient solution, the generation of a two N x N matrixes causes the program to become excessively bloated. Bench mark file ibm01.netD has approximately 12500 cells. An integer matrix of size, 12500 x 12500 is easily about 600 MB. Two of them will bring your total to 1.2 GB for two 2D vectors. Assuming that every vertex is connected to 20 other vertices, an integer 2D vector with 12500 x 20 entries is only 1 MB, which is 600 times smaller.

Implementing the 2D vectors found in generateG1 and generateG2 to have the form N x Y connections is a feasible implementation, but will require more time do to the need to search for the sink vertex for that particular source node. Due to time constraints, I was unable to create such a revised implementation for the generateG1 and generateG2 functions. Because of the bloated 2D vectors, I was not able to perform simulations using other ibm.netD files. For example, using ibm02.netD, the size of the 2D vectors created in generateG1 and generateG2 would each be approximately 1.4 GB. Creating such a vector caused the program to crash. My experience with MATLAB has been somewhat of a detriment when it comes to working with matrices in other coding languages.

Using small test files with a few vertices introduced some errors in the program. If the cut size was ‘1’ initially, the program would not realize that the optimal solution was already present. False starts were also present, where some cut size was presented, and the program would immediately proceed to calculate even large cut sizes afterwards, and then terminate when no improvement was found, with no minimal solution. These problems stem from the fact that we don’t expect to have a cut size of ‘1’ as either the initial cut size or as a relevant cut size at all. To ensure such errors don’t occur requires additional coding, which I didn’t find warranted the limited time I had to correct them when such a situation would never happen in a typical VLSI design.

During the initial trial of executing ibm01.netD, it took 6 hours to complete. I noticed the CPU usage was low for the program, as well as the CPU speed overall. Later in the run a large amount of memory allocated for the program was stored cached on the solid state drive. These factors lead to the dismal performance of the program initially. After checking my power settings on my laptop, it was set to power saver, which severally limits the maximum speed of the CPU. After shutting it off, CPU speed increased by four times. Coincidently I installed additional RAM into my laptop. This upgrade kept all the memory allocated for the program in RAM. The additional RAM and removing power saver mode boosted the computation time by four hours. Of course, introducing upgrades to hardware as a solution to speed computational performance is an extremely costly and wasteful practice. More efficient coding would have more of an impact than any additional hardware improvements.

IV. Experimental results

The ISPD98 Circuit Benchmark Suite [2] presents eighteen .netD files, containing from 13,000 to 210,000 cells. Due to unforeseen bottlenecks due to memory allocation, my program was only able to process ibm01.netD, which contains 12506 cells.

First, to ensure correct implementation of the code, three test files were created to ease debugging and to verify correct cut size calculation and partitioning though hand calculation. The following example, Figure 2, found in the textbook [4] correctly shows the minimum cut size to be '1'.

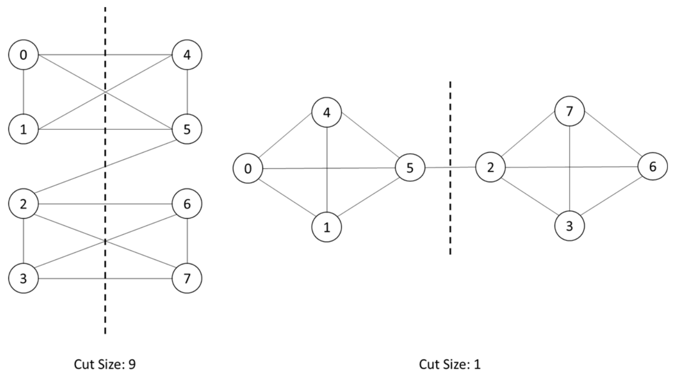

Figure 2: Textbook Example

Table 1 includes runtimes for one test file and one ibm.netD file. The runtime to find the minimum cut size of the graph in Figure 2, represented in textbook.netD was 5ms. Time to run ibm01.netD took just under 2 hours.

Table 1. Execution Time for .netD Files

File Name	**Initial Cut size**	**Final Cut size**	**Execution Time (s)**
textbook.netD	9	1	0.005
ibm01.netD	18176	5239	7347.21

It is puzzling to see that the cut size of ibm01.netD is in the thousands, as is seeing that the initial cut size is 18K, particularly because the number of nets in the file is around 14K, meaning there are 14K edges total, and the program found more than 14K edges.

The possibility of over counting edges crossed my mind. Perhaps the ibm files list both directions of the edge, but after searching through some parts of the file, I concluded that only one direction is given for all vertices. As for why the initial cut size is so high is a mystery. Additional time would be required to assess why this may have happened.

V. Conclusion

The implementation of the Kernighan-Lin algorithm presented in this paper has been proven to work correctly for small net lists, where hand calculation was possible to verify correctness. As for whether this implementation can correctly calculate the minimum cut size of the ibm.netD files found in the ISPD98 Circuit Benchmark Suite is currently unknown. Based on the smaller test cases run so far, the program should be able to carry out the K-L algorithm correctly. One probable culprit for the large cut size calculated for ibm01.netD could be due to incorrect parsing, with respect to avoiding vertices labeled as pads, 'p'.

The speed of the current algorithm is sufficient for the first ibm.netD file, but it is expected to increase with more cells. The bloating of the program due to the two N x N 2D vectors created by the program are highly inefficient for large VLSI designs, due to program crashes. Creating two N x Y vector where Y is the total amount of connected vertices for each i in N would allow for the execution of larger .netD files, although this would increase the total execution time by a small amount due to searching required to find the connected vertex for some source vertex, where $O(Y)$.

As the program winds down on the cut size, the calculation of the new cut size increases, due to more nodes being locked away after each execution, thus decreasing the amount of D and G gains that need to be recalculated.

The second part of the assignment asked to take area into effect when partitioning. Due to my long lapse in the use of C++ and various algorithm design and programming upsets, I was unable to entertain the idea of implementing such a feature in my program.

Although I was able to prove the ability of my program to execute the K-L algorithm on small test files, my poor choice in 2D vector implementation and possible misunderstanding of the structure of .netD files, I was unable to achieve sufficient benchmark characteristics for the ISPD98 Circuit Benchmark Suite ibm.netD files within the allotted time.

Acknowledgment

This paper on the implementation of the Kernighan-Lin algorithm in C++ was submitted as a project paper for ESE 556 - VLSI Physical and Logic Design Automation, conducted by Professor Alex Doboli at Stony Brook University.

References

[1] B.W. Kernighan, S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," Bell Systems Technical Journal, Vol. 49, No. 2, 1970, pp. 291-307

[2] C. Alpert. (201, Oct. 14). "The ISPD98 Circuit Benchmark Suite." Online.Available: http://vlsicad.ucsd.edu/UCLAWeb/cheese/ispd98.html

[3] C.M. Fiduccia, R.M. Mattheyses, "A Linear-Time Heuristics for Improving Network Partitions," Proceedings of the 19th Design Automation Conference.

[4] N.A Sherwani "Partitioning," in Algorithms for *VLSI Physical Design Automation*, 3rd ed. Norwell, K.A.P., 1999, ch.5