A Parallel FPGA Placer using GPUs

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Harish Nandan Mallapragada

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
Master of Science

Kiarash Bazargan

May, 2011

© Harish Nandan Mallapragada 2011
ALL RIGHTS RESERVED

Acknowledgements

I am very grateful to my advisor Prof Kiarash Bazargan for making me a part of his research group. His support and motivation were greatly helpful not only through the project but throughout my study at the graduate school. With out his timely guidance and inputs right from the problem formulation, this work would not have been possible by me. I would like to thank Prof. Sachin Sapatnekar for introducing me to the area of VLSI CAD through his courses. His courses on CAD were very helpful in understanding the various concepts of design automation. My special thanks to Dr. Weijun Xiao for the help and his time spent for discussing many problems regarding CUDA programming. His course on GPU computing at the university was very much helpful for programming in CUDA. I am thankful to Prof. David Lilja for allowing our work to be carried out on the machines in his lab. Thanks are due to Carlos Soria for his timely support with the many hardware issues we ran into. My thanks to everyone who have made my time at the graduate school so pleasant.

Dedication

To the Almighty

Abstract

With the increase in the complexity of circuits and decrease in the time-to-market, there has been a demand for more efficient and faster CAD tools. Placement forms one of the most critical stages in physical design. Simulated Annealing (SA) is widely referred to in the literature as providing the best quality solution to placement. In simulated annealing, millions of random moves are tried that change the location of the logic blocks with the goal of reducing a given metric - wirelength, area or delay - of the circuit. Hence longer execution time is the major drawback of SA. The parallel computing capability of Graphic Processing Units(GPUs), combined with their low cost has made GPUs a favorable choice for a wide range of high performance computing applications. The main challenge in using GPUs is the limited amount of available memory. In this work we propose methods to carry out the random moves of SA in parallel with the help of multiple GPU threads. We show methods to partition the available memory to minimize the interaction between the sub problems thus maintaining the accuracy of the serial algorithm.

Our parallel placer is based on Versatile Placer and Router(VPR), an industry standard SA based placer. Our method works for circuits with up to 1500 logic blocks and performs 37% faster compared to VPR with a maximum increase of 17% in the required wirelength.

Contents

Acknowledgements **i**

Dedication **ii**

Abstract **iii**

List of Tables **vi**

List of Figures **vii**

1 Introduction **1**

2 FPGAs and FPGA CAD **3**

2.1 Architecture of FPGAs 3

2.2 FPGA CAD 5

3 Placement for FPGAs **7**

3.1 Classification of Placers 8

3.1.1 Partitioning based Placers 8

3.1.2 Analytic Placers 8

3.1.3 Simulated Annealing based Placers 8

3.2 The Simulated Annealing Algorithm 9

4 Prior work **13**

5 The parallel Placement **16**
5.1 Overview of the VPR's Placer . . . 17
5.2 Comparison of Possible Parallel Approaches . . . 19
5.3 The Proposed Method . . . 19
5.3.1 Challenges Involved . . . 20
5.3.2 Avoiding the thread clashes . . . 20
5.3.3 Handling the data structures . . . 24
5.3.4 The Algorithm . . . 24

6 Results **33**
6.1 Comparison of Placements . . . 33
6.2 Analysis of Results . . . 34
6.2.1 Synchronization Overhead . . . 35
6.2.2 Compute to Memory Access Ratio (CMA) . . . 35
6.3 Comparison of Routing of the Placed Circuits . . . 38

7 Conclusion and Future Work **47**

References **49**

List of Tables

5.1 Profiler Output on VPR 17
6.1 Comparison of speed-up between different thread configurations 34
6.2 Comparison of synchronization overhead between different thread configurations 35
6.3 Comparison of final bounding box values between different thread configurations 43
6.4 Comparison of quality metrics between different thread configurations . 44
6.5 Degradation in quality metrics by the parallel method 45

List of Figures

5.1 Reduction with thread Divergence 22
5.2 Reduction without thread Divergence 23
5.3 Conflicts between the parallel threads for e64-4lut.net with 404 logic blocks 25
5.4 Conflicts between the parallel threads for apex4.net with 1500 logic blocks 26
5.5 Acceptance rate in the parallel version 27
5.6 Acceptance rate in the serial version 28
5.7 Acceptance rate with 3 times more number of moves per temperature step 29
5.8 Acceptance rate with 9 times more number of moves per temperature step 30
5.9 Acceptance rate with 64 times more number of moves per temperature step 31
5.10 Acceptance rate with 256 times more number of moves per temperature step 32
6.1 Annealing with *block* in the shared memory 37
6.2 Annealing with *clb* in the shared memory 38
6.3 The serial annealing schedule 39
6.4 Annealing with 4 parallel threads 40
6.5 Annealing with 8 parallel threads 41
6.6 Annealing with 16 parallel threads 42
6.7 Annealing with 32 parallel threads 43
6.8 Annealing with 64 parallel threads 44
6.9 Annealing with 128 parallel threads 45
6.10 Annealing with 256 parallel threads 46

Chapter 1

Introduction

The popularity of FPGAs is mainly due to the faster time-to-market than the ASICs. In order to match the fast growing needs and circuit sizes, there is a huge demand on the CAD tools for FPGAs to be faster. Many FPGA placement tools in market today use Simulated Annealing(SA) based approaches in which millions of random moves are tried changing the location of the logic blocks with the goal of reducing a given metric - wirelength, delay, area - of the circuit. Due to the low cost and parallel computing capability of Graphic Processing Units (GPUs), GPU computing has recently been used for a wide range of high-performance computing applications. In order to overcome the long run times of the SA, we propose to use GPU threads to attempt the random moves in parallel. In this work we show methods to efficiently utilize the power and available memory on the GPU to speed-up SA, yet maintaining the accuracy of the serial algorithm.

The rest of the chapters are organized as follows.

- Chapter 2 briefly describes the basic architecture of FPGAs and the current trends in FPGA CAD tools.
- In Chapter 3, we discuss the problem of placement and different types of placement in greater detail.
- Chapter 4 describes the previous published works on parallel placement and particularly simulated annealing based methods.

- In Chapter 5, we introduce the Versatile Place and Route (VPR) tool and our enhancements to it to develop our GPU based parallel placer.
- In Chapter 6, the simulation results of our work in comparision to the existing tools is presented.
- Chapter 7 concludes this work and discusses the future scope.

Chapter 2

FPGAs and FPGA CAD

With the increasing competition in the electronics industry, getting the product out into the market in the shortest time possible has become extremely vital. Moreover, designers' innovative ideas are limited by the financial risk involved in the development of a new product. Introduced in 1984, Field Programmable Gate Arrays (FPGAs) have become popular because of two key advantages, lower non-recurring engineering (NRE) costs, and faster time-to-market. Other implementations of digital circuits like Mask Programmable Gate Array (MPGAs) require that a new chip is fabricated to implement a single design. Apart from the high NRE cost which covers lithography and the cost of running the first chip through the fabrication plant, the time taken to get the first chip working is typically about 6-8 weeks. This process is not worth the cost and effort unless large volumes of chips are produced. Moreover, any slight deviation in the functionality of the chip requires the whole fabrication process be repeated, further increasing both cost and time-to-market. On the other hand, implementation on an FPGA is just a matter of programming and any bugs found after testing, can be corrected in minutes[1].

2.1 Architecture of FPGAs

FPGAs in general are made of three fundamental components: I/O blocks, Logic blocks and Programmable routing. Each logic block implements a small portion of the logic required by the circuit while each of the I/O blocks serves as an input or output pad for the circuit. The (programmable) routing can be configured to make necessary connections

between the logic blocks.

Logic Blocks: An FPGA consists of many logic blocks arranged in either 2D array or a row-based fashion and all of these logic blocks collectively implement the functionality of the circuit. Depending on the architecture of the FPGA, a logic block can be as simple as a two input NAND gate or be a more complex element like a multiplexer, Look Up Table(LUT) or even a PLA in some complex architectures.

The performance of the FPGA - the area occupied by a given circuit, the critical path delay - is highly dependant on the logic block architecture. Apart from directly affecting the area, the logic block architecture also affects the area occupied by the routing resources. A simple and small logic block occupies less area but the interconnect area required to connect these blocks increases. Apart from adding to the area, routing resources also increase the delay of the circuit. On the other hand, bigger logic blocks directly add to the area of the FPGA. [2] has shown that excessive number of pins on a single block increase the total chip area.

In [2] it is shown that the best tradeoff between the performance and area on an FPGA can be achieved by increasing the functionality of each pin a logic block. This can be best achieved by using logic blocks based on Look Up Tables(LUTs). A 3 input LUT can implement as many as 256 logic functions of its 3 inputs. Moreover, it is also shown that the number of inputs k of a k-input logic block is to be chosen as 4 for optimal performance. Most FPGAs commercially available today are LUT-based. Logic blocks in these modern FPGAs are composed of clusters of LUTs connected by local interconnect and some registers in between them. Various varieties of architectures have been proposed based on the groups of LUTs approach. A combination of two different sized LUTs has been studied in [3]. [4] proposed groups of 4-LUTs in a logic block with hard wired interconnect.

Hence most benchmark architectures we used to test our placer are based on logic blocks with groups of LUTs. However the number of LUTs in each logic block and their internal architecture vary between each of the architectures we used.

Programmable Routing:

The routing architecture of FPGAs includes wire segments and programmable switches. The programmability of FPGAs is based on switches -multiplexers, pass transistors or tri-state buffers- which are controlled by SRAM cells in most of the current devices.

Examples of SRAM based FPGAs include most of the FPGAs from Xilinx [5], Altera [6] and Lucent [7].Other technologies than SRAMs are based on antifuses (many of the Actel Devices [8]), EPROMs and EEPROMs.

The area requirement for SRAM based programming technology is higher due to the additional transistors required by pass transistors or transmission gates. Also, the RAM being volatile, the chip requires some sort of storage for the RAM cell bits. On the other hand it has the advantage of being produced by a standard CMOS process and also in-circuit reconfigurability. EEPROM cells do not require this external storage but are not in-circuit reconfigurable. The area required by anti-fuse based programming technology is less but their manufacture is not as easy as a basic CMOS process.

2.2 FPGA CAD

FPGA Design Flow

Most commercial FPGA manufacturers provide the CAD tools specific for their devices for synthesis and analysis of the user's designs. Each family of devices has its own defined design flow. The most general steps in FPGA design flow are described in this section.

Design entry The user's design is read into the tool in this step. The design is described either as a HDL file or a schematic. The design description for some commonly used components like adders, counters etc. are provided in most design entry tools. Tools like Xilinx ISE [9], provide interface to define system clock graphically.

Synthesis The synthesis tool generates a netlist from the given HDL description. It can be divided into 3 broad steps.

1. Check Syntax: The tool checks the HDL source code for any syntax errors.
2. Compile: This step translated the HDL code into a netlist which consists of generic cells.
3. It is in this stage that the components from the previous stage are translated to the components in the target technology's library.

The user can opt to modify the way the synthesizer performs by giving options. For

example, the maximum fanout of a flip-flop or maximum area etc. Note that the synthesis step is not required if the user chooses to enter a schematic design. In such a design entry, the components themselves come from the device's technology library.

Behavioral Simulation The designer can choose to check the functionality of the HDL or schematic design using a simulator. This is usually done using a HDL testbench. Though this step is optional in the design flow, this greatly helps designers to address any remote bugs before the design is actually implemented.

Placement Each logic cell from the mapping step is assigned a specific location on the FPGA. this is done either by the tool or manually. We will be discussing more on placement in the subsequent sections.

Routing After the cells of the circuit are mapped to the logic blocks on the FPGA, the programmable switches in between the wire segments are to be configured. This is typically done by the routing tool, though sometimes further manual optimization by the designer is preferred.

Post-Map Timing Analysis The behavioral simulation step, though it gives the timing of the circuit is not as accurate as the placement of the cells and the wire segments used in the routing were uncertain. The timing analysis step after the mapping gives accurate timing report of the design.

Configuration data The analyzed design is used to create configuration data which will be downloaded to the target device.

Chapter 3

Placement for FPGAs

The placement step in the design flow determines a logic block on the FPGA for each of the logic blocks of the circuit. In other words, the circuit's blocks are mapped to the FPGA's logic blocks. Based on the requirement of the circuit, the physical location of each of the blocks of the circuit is determined and the placements [1]are classified as:

1. Wirelength Driven : Logic blocks that are connected are placed close to each other in order to reduce the total wirelength of the circuit.

2. Routability Driven : The blocks are placed so as to balance the utilization of wiring resources across the FPGA. Given the limited space for routing in an FPGA as opposed to an ASIC, routability driven placement is preferred for most applications.

3. Timing Driven : The blocks may be required to be placed such that the speed of the circuit is maximized.

Different algorithms proposed for the placement problem can be broadly classified into partitioning, analytic or simulated annealing based placers.

3.1 Classification of Placers

3.1.1 Partitioning based Placers

A partitioning based placer divides the given circuit of n cells into two sub-circuits of n/2 cells each, then these two divisions into further smaller sub-divisions and goes on in a binary tree fashion until there are only few cells remaining in each of the sub-division. Partitioning is done under the constraint that the sum of the edge weights between two sub-divisions is minimum when they are partitioned. This ensures that closely related cells are retained in the same groups. In the final layout all cells in a given group are placed close to each other. This method [10] proves to yield areas within 10-20 percent of that from a hand placement. In [11], a partitioning-based placer which makes use of the circuits delay estimations -obtained from the routing profiles of selected circuits, which have already been placed and routed- has been proposed. This method where the placer is aware of the routing delays, yielded better post-routing circuit delays.

3.1.2 Analytic Placers

Analytic placers [12] [13] formulate the placement problem as a set of quadratic equations which describe the location of cells and connections between them. These quadratic equations are solved under certain constraints. As opposed to partitioning based placement, analytic placement does not depend on an initial placement. [14] proposed to introduce a linear objective to the quadratic programming approach of analytic placement.

3.1.3 Simulated Annealing based Placers

The simulated annealing based placers avoid stucking at the local minima. The term annealing refers to the process of slow cooling of molten metals to form properly shaped solids. The logic blocks are allowed to occupy any location over the area of the entire chip in the initial stages (analogous to the liquid state at very high temperature). At these high temperature stages, even the moves which increase the cost of the placement are also accepted. Later as the temperature decreases, the moves which increase the cost function are selected but with a reduced probability (analogous to reduced mobility

of particles at lower temperatures). This property of accepting the hill climbing moves helps this algorithm to avoid stucking at local minima.

3.2 The Simulated Annealing Algorithm

Before we describe in detail about the simulated annealing algorithm, we define a few key terms from the above paragraph which will be used throughout the discussion.

The term *cost* refers to any function that determines the quality of the final placement. For example, in a routability driven placement, the cost function is defined in such a way that the slots around narrowly routed channels are kept at high cost. This ensures that the parts of the circuit that require high wiring are placed at those areas where congestion is not a major issue. Another example of placement cost is the sum of the wirelengths over all the nets in the circuit.

A *random move* is typically swapping the on-chip locations of two randomly selected logic blocks of the circuit. This in turn changes the length and shape of all the nets connecting these blocks and hence the placement cost.

The change in the total placement cost due to a particular move is what determines it to be a *good* or a *bad* move. A good move is that which reduces the cost of the placement and this is always accepted whereas a bad (cost increasing) move is accepted with a probability which is dependant on the current temperature.

The *temperature* is a parameter which is used to determine whether a particular move is to be accepted or not. This is set high initially and gradually reduced as the annealing proceeds. The higher the temperature, the more freely the blocks can be placed.

The most general steps a simulated annealing engine follows can be outlined as below

1. Make an initial random placement. Set the temperature very high.

2. Pick a pair of logic blocks randomly and swap them. Calculate the change in the total cost due to this disturbance.

3. If it is a good move keep it. If it is a cost increasing move decide *depending on the temperature.*

4. Repeat steps 2 and 3 for a large number of times before reducing the temperature.

5. Repeat steps 1 through 4 until temperature reduces to zero.

6. This process terminates when a desired quality is achieved or the temperature reduces to zero.

The rate at which the temperature is modified, the cost function, the number of moves at a given temperature determine the quality of the placer. The faster it attains a lower cost, the better the annealing schedule is. Various variants of simulated annealing have been proposed based on the choice of initial temperature, cooling schedule, and the exit condition etc. Various types of annealing algorithms have been published in the past.[15]-[16] give some of the prominently known variations of simulated annealing.

The VPR (Versatile Placer and Router) [17] is a placement and routing tool for array based FPGAs. It is named versatile in the sense that it targets FPGAs of various architectures. The placer in VPR is based on simulated annealing but with many enhancements which will be described later in this section. Due to the versatility and wide variety of applications of VPR's placer, we chose it to implement on the GPUs and verify our parallel algorithm.

Since VPR supports a wide variety of architectures, an FPGA is described in an argument file which specifies the dimensions of the array of logic blocks, the architecture of the basic blocks - the number of inputs and outputs and their directions-, the number of I/O pads etc. It also describes the width of the routing channels at different areas of the chip. The circuit netlist format for VPR describes the circuit as a group of logic blocks and nets connecting these logic blocks. In other words, the architecture file gives the legal slots on the chip where each of the logic blocks of the circuit can be placed. The problem of placement an now be defined as mapping the logic blocks of the circuit

to the slots in the FPGA while satisfying the given constraints.

We now briefly describe the prominent features of the VPR's placer which we have maintained in our implementation.

To choose the initial temperature, VPR uses a schedule similar to that used in [18]. N number of moves are evaluated on an initial random placement and the standard deviation of the costs (sigma) of these moves is calculated. The initial temperature is set at (20 * sigma). This ensures almost all the moves being accepted during the initial stages.

The number of moves attempted at a given temperature is also kept dynamic in VPR. As shown in [19] the number of moves is made dependent on the number of blocks in the circuit.

New Temperature Updating Schedule

As seen in the previous section, in the initial stages of the schedule almost all the random moves are accepted including those which are bad. This reduces the scope for cost improvement. During the final stages - the low temperature stages- the schedule becomes more selective and very few moves are accepted thus again reducing the chances for cost reduction. Therefore, the middle stages where most but not all moves are accepted are the ones where cost improvement is achieved. Hence in VPR, the temperature update schedule is made such that much of the time is spent in these most productive stages.

$$t_{new} = \gamma \times t_{old} \tag{3.1}$$

where γ depends on α, the number of moves accepted at t_{old}

A new temperature t_{new} is calculated from the current temperature t_{old} depending on the fraction of accepted moves α at the current temperature. The *less productive stages* are those where most or least number of moves are accepted, that is α is either 1 or 0 respectively. At these stages, γ is kept significantly smaller, ensuring that temperature is updated very fast. γ is kept very high in the middle stages where α is about 0.44. This helps the schedule proceed very slow exploring many temperature steps at these stages.

The VPR gives a choice of three different cost functions for placement.

The linear and non-linear congestion costs fall into the category of routability driven

placement. These take into account and model the effect of non-uniform channel widths over different regions of the chip.

The bounding box cost used for most of the FPGA architectures is defined as the sum of the perimeters of the bounding boxes of all the nets in the circuit. Simulated annealing is carried out with the aim of reducing this bounding box cost of the nets. Due to the wide applicability of this cost function we chose to try the current and first version of parallel algorithm on this cost function.

In spite of the quality of placement achieved by the simulated annealing based placers, the inherent drawback of this algorithm is very low speed. The quality of the final result is directly dependent on the number of random moves made at each temperature step and the number of temperature steps visited. Hence in the recent years much focus has been on methods to speed-up the simulated annealing. This gave rise to many approaches which attempt to parallelize the various parts of the algorithm. Chapter 4 describes several published approaches to parallelize simulated annealing. The method we adopted in this work is detailed in chapter 5.

Chapter 4

Prior work

In general variations of four approaches to parallel simulated annealing are followed.

- The parallel moves approach in which the moves are generated independently by all the processors and evaluated. Of those that are accepted, the best one is chosen and this is set as the initial condition for the next iteration. [20] proposed to use different parallelization techniques for high and low temperatures of annealing. They have also shown that this method is particularly suited for higher temperature stages. The problem with this approach is that no processor can see the moves of any other processor and there are high chances of errors in the cost evaluation. Different methods have been proposed in the literature to address this problem.

- In area based partitioning the FPGA is divided into parts and each processor is given the monopoly on each of the partitions. Any processor can move any net as long as the net is completely in its partition. Variations of this method are shown in [21] [22] [23]. While this approach overcomes the error in cost, it is limited in the sense that the moves are too restricted.

- The synchronous Markov chains approach has also been widely applied in which each processor is allowed to carry on a chain of moves independently until a specified time when all the processors report their costs. The best placement so far is now made the initial condition for the next chain of moves by the processors. [24] [15] [16] show different methods based on synchronous markov chains.

- In the Asynchronous Markov chain approach processors do not wait for all other processors to complete before synchronizing. Each processor queries a server at regular intervals and copies the placement either to the server or from the server depending upon which is lower. Thus the synchronization overhead is removed. [24] has compared and contrasted this markov chain approach with the synchronous markov chains method and concluded that the asynchronous approach worked best.

The Markov chain approach has been applied to parallel VLSI cell placement in [15]. A speculative computation which looks one step ahead has also been verified in the work. Though the Markov chain method showed some speedup over the pure parallel moves approach, the speculative algorithm actually gave negative results which have been attributed to some inherent characteristics in the placement like relatively high acceptance rate, large state size etc.

[22] used three proposed parallel simulated annealing algorithms on the Timberwolf placement tool to develop their parallel placement tool Properplace. The area partitioning approach used in properplace is different from others in that the circuit is divided into rows and each row is given to a different processor which reduces the error in the cost calculation due to cell overlaps.

[21] implemented the parallel moves approach with a little change that a single processor generates the moves while all others evaluate each of them thus overlapping generation and evaluation of moves. They conclude that the reduction in the synchronization frequency gave the best results in terms of the speedup.

In [20] at lower temperatures an adaptive algorithm decides the number of processors allocated to the parallel machine and the number of moves carried out by each processor between synchronizations, while at higher temperatures parallel evaluation of independent chains of moves has been adopted. But this algorithm degrades the quality for smaller circuits.

[25] attempted to parallelize simulated annealing for room assignment problem on shared and distributed memory platforms and concluded that synchronous move generation using message passing works best on a multicomputer platform whereas for a shared memory, multi threaded model based on openMP parallel loops is suggested.

[26] created a function to show how quality of an answer by parallelization is compensated relating it to the number of iterations. They adopted a temperature update procedure which is constrained by maintaining a quasi equilibrium. That is, move generation strategies have an impact on the temperature.

[16] proposed a speculative algorithm with 3 processors out of which one performs the sequential algorithm and the remaining two perform speculative sequential operations on the possible outcomes of the first. It showed a logarithmic speed up depending upon the number of processors utilized.

Though not specific to the problem of placement, [27] [28] [29] [30] show various parallel approaches to SA targeted to different applications.

Chapter 5

The parallel Placement

The implementation of our algorithm is based on the serial version of VPR[17]. We made the necessary changes to the C source code and inserted CUDA modules as and when necessary. Since the main motivation is to speed up the implementation, analysis of the processor time on each part of the algorithm is essential. Table 5.1 shows the output of profiling done on the source code.

Flat profile:

Each sample counts as 0.01 seconds.

Of the many modules in the placement program, as can be expected the module "try-swap" which tries different placement states and compares them consumes the most amount of time. Hence for our parallel implementation, we chose this module to start with.

Implementation of an essentially serial algorithm in parallel involves two main challenges. First the inherent data dependencies and the second is managing the data elements in the available memory. Given the different levels of memory on the GPUs and the large variation of access times between these levels of memory, memory utilization and mapping of different data structures to these memory spaces forms the critical part of the GPU implementation. This makes the analysis of the data structures used and the accesses to these structures in the serial implementation very significant. In this subsection we briefly give the details of the data structures and functions in this part of the code.

Table 5.1: Profiler Output on VPR

% time	cumulative seconds	self seconds	name
64.53	3.73	3.73	try_swap
10.21	4.32	0.59	comp_td_point_to_point_delay
8.82	4.83	0.51	update_bb
6.75	5.22	0.39	get_bb_from_scratch
5.02	5.51	0.29	my_irand
2.25	5.64	0.13	try_place
1.04	5.70	0.06	my_frand
0.17	5.71	0.01	check_rr_graph
0.17	5.72	0.01	empty_heap
0.17	5.73	0.01	get_heap_head
0.17	5.74	0.01	get_rr_node_index
0.17	5.75	0.01	get_ytrack_to_clb_ipin_edges
0.17	5.76	0.01	get_ytrack_to_xtracks
0.17	5.77	0.01	load_rt_subtree_Tdel
0.17	5.78	0.01	timing_driven_route_net

5.1 Overview of the VPR's Placer

VPR handles the netlist data by three main data structures.

Net: This data structure describes each of the nets in the circuit using these parameters
Name of the net in ASCII
Number of pins on this net
List of all the blocks that this net connects to

Block: Each of the logic blocks in the circuit is described by this data structure.
Type of the element (Input, Output or CLB)
List of nets connected to this block. This entry is let blank for an unconnected pin
Physical location of this block in terms of an ordered pair (x,y)

CLB: The legal slots on the FPGA are described as a 2-D array of this basic unit
Type of the element occupying this slot (Input, output or CLB)

Number of logic blocks mapped to this slot (Ideally it should be either 1 or 0)
Number of the logic block occupying this slot

The text processing modules in the VPR process the input netlist and the architecture files and represent all the required data in the above three data structures. In order to carry on the function by parallel threads, a detailed analysis of all the sub-functions is of importance. The main steps performed by the placer are as follows

Initially set temperature and also the range limit *rlim* high. The variable *rlim* determines the maximum distance a block can be moved.

1. Generate a random number between 1 and N_{blocks}. The block with this number is one of the blocks to be swapped Lets call that *b_from*.

2. Generate two random numbers x_{to} and y_{to} between 1 and rlim. The position (x_{to},y_{to}) will be the new location for the block b_{from}.

3. The location (x_{to},y_{to}) is looked up in the clb array and the block b_{to}in this location is the block to be swapped with.

4. Each of the nets connected to both these blocks is examined. The bounding box of a net which is connected to both these blocks is not affected by swapping these blocks whereas the wirelength of the nets connected to either of these two blocks need to be recalculated.

5. For each of the nets which are disturbed by this move, the bounding box is calculated by traversing all the blocks that net is connected to and recording their locations. Note that, the location fields of both b_{from} and b_{to} blocks are already swapped and this gives a different wirelength for the net as compared to the original state. The new wirelengths are compared to the old ones of the respective nets and the "delta" cost is added up over all the nets.

6. This "delta" cost is given to the Boltzmann function which decides to accept or reject this move depending upon the current temperature.

7. Depending upon the outcome of step 6, the move is either committed or rejected. Note that till now except the block, no other data structure has been disturbed. For committed moves the other data structures are updated. For rejected moves, the moved blocks are set at their respective initial positions.

8. Repeat steps 1 through 7 for $N_{blocks}^{4/3}$ times.

9. Reduce the temperature depending on the number of successful moves and repeat all the above steps(only if the exit condition is not met).

5.2 Comparison of Possible Parallel Approaches

The above analysis suggests two basic ways of parallelizing the algorithm.

1. Divide the tasks among threads and dedicate each thread to a particular stage of the algorithm so that the process is speeded up.

2. Allow all the threads to go through the complete algorithm so that only $N_{blocks}^{4/3}/N_{threads}$ number of moves are required at a given temperature step as opposed to $N_{blocks}^{4/3}$ number of moves in the serial version thus speeding up the process by $N_{threads}$ times.

Though the first method seems to be a reliable way of speeding up any process, the inherent data dependency of different stages in the simulated annealing algorithm makes the application of this process more challenging. For example, unless the nets which are disturbed are recognized, the change in cost of those nets can not be determined. This approach to parallelization can be found in [21] [20]

Another major drawback of this approach, particularly when implemented on GPUs is the overhead of memory transfers between the host CPU and the GPU which forms the major part of the time consumed by the implementation. More details on memory transfers are discussed in subsequent sections of this chapter.

5.3 The Proposed Method

Considering the massive parallelism of the GPUs and the problem of divergence discussed in the previous section, in this work we chose the second approach -to distribute

the same work among all the threads and reduce the number of iterations executed by each thread- to speed up the placement process. This method shows a clear advantage of avoiding the divergence problem.

5.3.1 Challenges Involved

We now enumerate the issues that arise in this approach. Our methods to address these issues follow.

1. When two or more threads try to pick two (random) blocks from the circuit and swap their locations, there is a probability that they pick the same block to move to two different locations. Conversely, they may try to place two different blocks at the same location of the chip.

2. The data structures which store the circuit information, are no longer guaranteed to be correct as they might have been overwritten by some other thread.

3. Without each thread knowing the moves of all the other threads, the accuracy of the calculated cost function might not be reliable.

4. The update scheme for the temperature, which depends on number of accepted moves at a given temperature, behaves differently for each of the threads thus creating divergence among them.

5.3.2 Avoiding the thread clashes

To address the problem of multiple threads trying to move the same block, we developed a reduction module that is run in parallel by all the threads. The following pseudo-code shows the functionality of the reduction module.

```
add_index = 2;
div = 1;

while(div<=num_threads)
{
if(threadIdx.x % div == 0)
```

```
{
for(int i=0;i < num_blocks;i++)
c_array[threadIdx.x][i]=c_array[threadIdx.x][i] || c_array[threadIdx.x + add_index][i]
||(u_array[threadIdx.x][i] & u_array[threadIdx.x + add_index][i]);

__syncthreads();

for(int i=0;i < num_blocks;i++)
u_array[threadIdx.x][i]=u_array[threadIdx.x][i] || u_array[threadIdx.x + add_index][i];

__syncthreads();


}
div = div*2;
add_index = add_index*2;
__syncthreads();
}
```

Reduction

Each thread has its own copy of two arrays *u_array* and *c_array*. *u_array* (utilization array) contains a record of block numbers chosen by each of the parallel threads. Every second thread (thread Ids 0,2,4....etc.) in the schedule calculates the block conflicts between itself and its neighbor. A bit in the conflicts array *c_array* corresponding to a particular block is set if it has been picked by 2 or more threads. In the next iteration every fourth thread (thread Ids 0,4,8..etc.) repeats the above operation and updates its conflict array. Thus after *n*log n* number of iterations, *c_array* thread 0 contains the information of all the block conflicts in the schedule. Both the *c_array* and *u_array* are placed in the shared memory so that each thread can see the blocks picked by every other thread.

The operation of the above described method is shown in the figure 5.1 with a simple example. Observe that when we run this module on more than 32 threads, the warp divergence problem arises since two consecutive threads are performing two

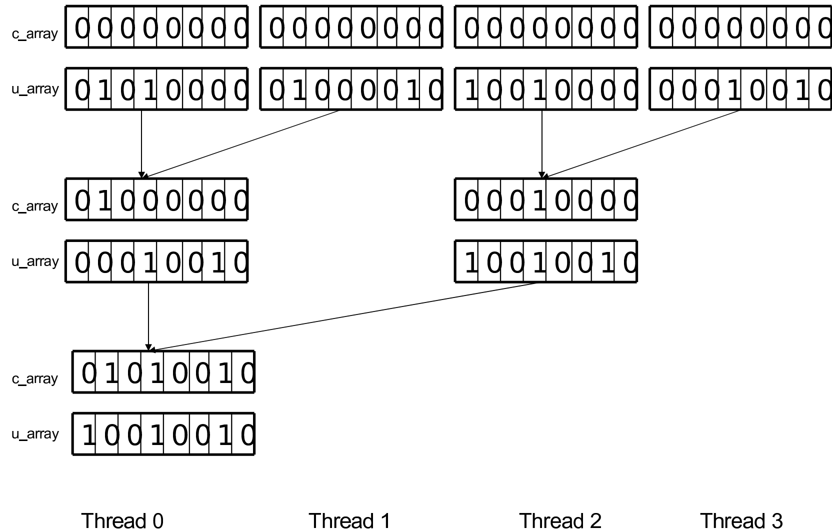

Figure 5.1: Reduction with thread Divergence

different operations. Thus we modified the above algorithm such that threads belonging to the same warp perform the same operation at most possible times. This is illustrated in figure 5.2. The solution without divergence clearly shows an advantage over the previous one. Another advantage of the reduction approach is that both the utilization and conflicts arrays are of type *boolean* occupying only one bit which saves significant amount of shared memory.

Though the parallel approach for reduction effectively eliminates block clashes, it is severely limited by the size of the shared memory. To perform reduction, each thread needs to keep track of the blocks picked up by other threads. This requires each thread to have its own copies of both *c_array* and *u_array* in the shared memory. Keeping in view the 16 KB of shared memory on the Tesla GPU, it is not practically possible to go beyond 16 threads for the benchmark *e64-4lut.net* with 404 blocks and 339 nets. It is even more limited for bigger circuits. For example, only 8 threads could be run for *apex4.net* benchmark with 1500 logic blocks.

To avoid the excessive use of shared memory, and also to be able to run more number of parallel threads for bigger circuits, we use a serial counter which counts the number

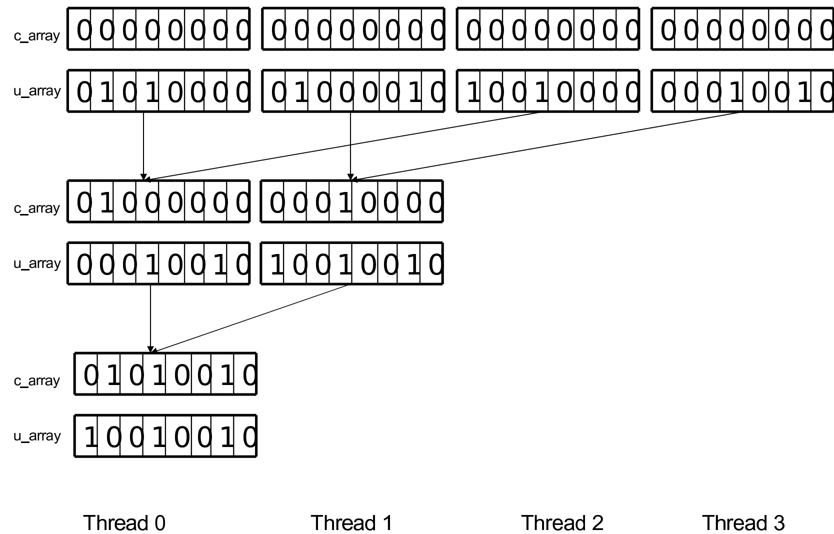

Figure 5.2: Reduction without thread Divergence

of threads acting on each block. This counter is updated by a single thread. This simple counter is shown below.

{Each thread writes the numbers of the blocks it wants to swap to this array}
$block_picked$[threadIdx.x] $= b_from$
$block_picked$[threadIdx.x $+ num_threads$] $= b_to$
$__syncthreads()$
{Wait until all threads have finished swapping}
{Only one thread is allowed to do the following operation}
if $threadIdx.x = 3$ **then**
  for $ptr = 0 \to 2 \times num_threads$ **do**
    $num_of_times_picked[block_picked[\text{ptr}]] = num_of_times_picked[block_picked[\text{ptr}]] + 1$
    $__syncthreads()$
  end for
end if

The barrier synchronization before starting the counter is essential to ensure the correct operation. The advantage of allowing a single thread to modify the counter is that one global copy of the counter is sufficient in the shared memory unlike the previous method where each thread has its own copy of the conflict and utilization arrays. As shown in the results section, the effect of the serial counter on the speed is negligible when compared to the speed-up obtained by the extra number of threads that this method can run.

5.3.3 Handling the data structures

As mentioned earlier in this chapter, VPR handles the netlist and architecture information using three main data structures *s_block*, *s_net* and *s_clb*. The data accesses are the highest to these data structures throughout the process of annealing. We keep these data structures in the global memory so that any thread can read the data from them.

5.3.4 The Algorithm

For each data structure the serial version uses, we maintain an additional dummy structure which is local to each of the threads. In the above mentioned serial algorithm, all the steps from 1 through 7 are carried out by each thread independently on its local data. That is, each thread does the following steps independent of others

1. Look up the global data structures and copy them into its local (register) memory.
2. Choose the two blocks and their locations to be swapped.
3. Record these values in the utilization array *u_array*.
4. Swap them in its own local copy of data.
5. Calculate the changes in bounding box of each net throughout the circuit.
6. Do the temperature test on the move just made.
7. Set the flag if the move is accepted.
8. Synchronize with all the other threads.

After performing these steps each thread is stopped by a barrier synchronization until all the other threads reach this point. At this point a single thread updates the conflicts array. This array shows which blocks in the netlist have been chosen by more than one thread. Figure 5.3.4 shows a typical number of thread conflicts that arise when 256 threads are making parallel moves for 500 times at any given temperature.

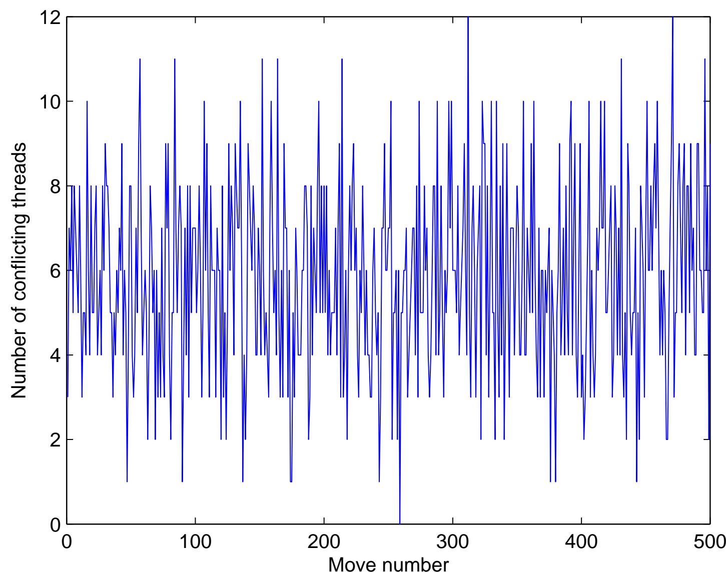

Figure 5.3: Conflicts between the parallel threads for e64-4lut.net with 404 logic blocks

It can be observed in figure 5.3.4 that the number of thread conflicts is smaller in a larger circuit as there are more number of blocks for the threads, to choose from.

In the event of two threads picking the same block, we discard the moves made by both the threads. The effect of these wasted moves is that our acceptance rate is slightly deviated from that of serial VPR. Figures 5.3.4 and 5.3.4 compare the acceptance rates of our method to the original version for *e64-4lut* circuit.

The reason for the deviation in our acceptance rate is the block clashes between the threads. Apart from the temperature, there is an additional random factor that checks our moves. Due to this, our acceptance plots are not as smooth as the serial version.

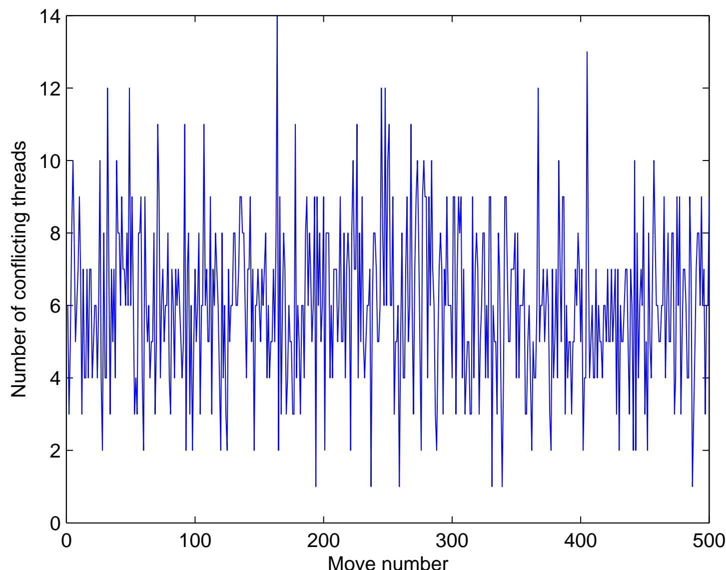

Figure 5.4: Conflicts between the parallel threads for apex4.net with 1500 logic blocks

By increasing the number of random moves made at each temperature step, the probability of moves being wasted -due to block clashes- reduces, giving higher acceptance rates. Figures 5.7 to 5.10 show the acceptance rate curves with different number of *moves per temperature* with 256 threads. It can be seen that as the number of moves made at a given temperature increases, the curve tends to resemble that of the serial version. But this comes at the cost of additional runtime. For instance, when the number of moves per temperature is increased by 3 times, the runtime also showed an increase of 3 times as compared to our original parallel implementation (in which we make equal number of moves per temperature as that in the serial version).

The temperature and the range limits are then updated by a single thread based on the accepted number of moves in this iteration. All threads now repeat the above process independent of each other with the updated value of temperature. The program returns to the host CPU after the exit criterion is met. All the intermediate values like the temperature steps, costs etc. are all stored in appropriate data structures and are returned to the host at the end as there is no way to interact with the GPU while the kernel is being executed.

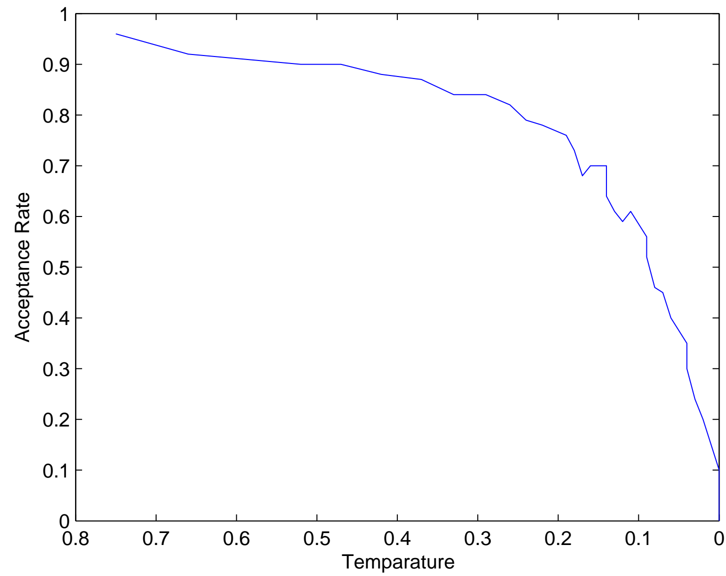

Figure 5.5: Acceptance rate in the parallel version

The performance results of the parallel algorithm as compared to the original VPR both in terms of quality and speed are shown in the following chapter.

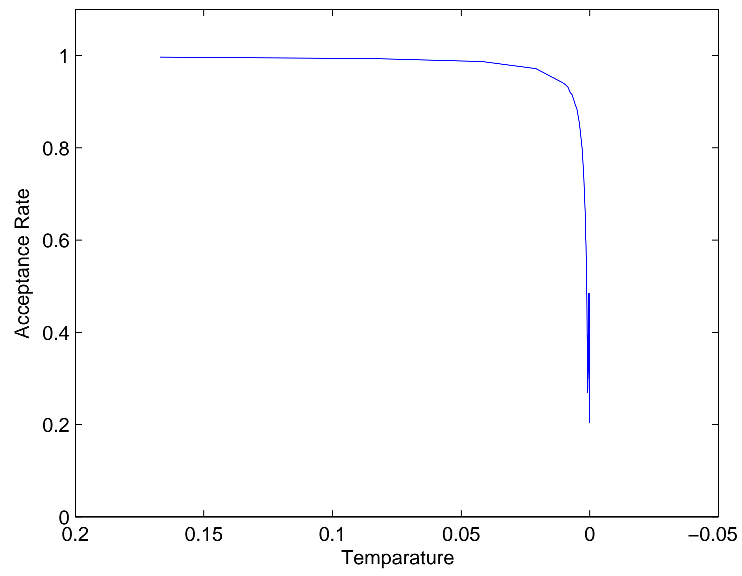

Figure 5.6: Acceptance rate in the serial version

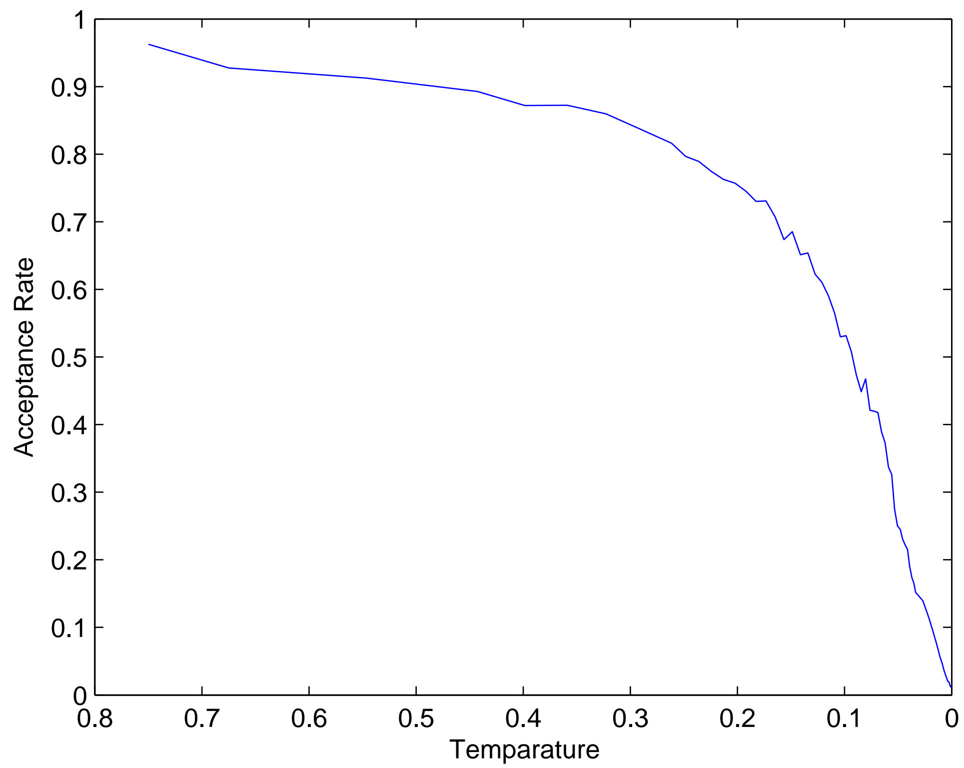

Figure 5.7: Acceptance rate with 3 times more number of moves per temperature step

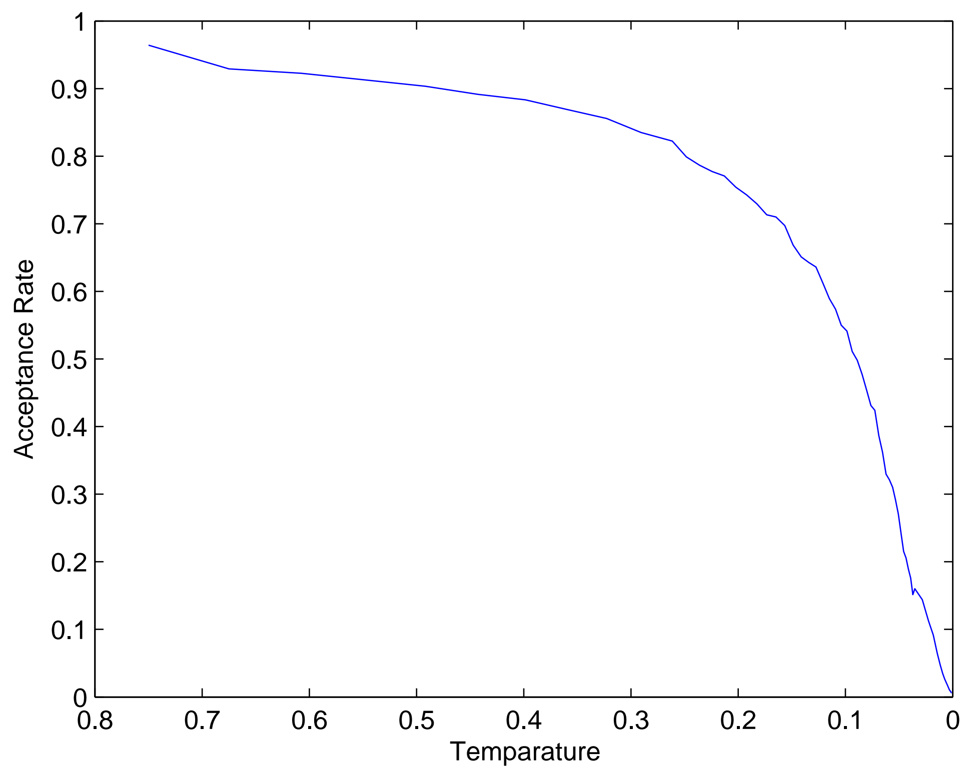

Figure 5.8: Acceptance rate with 9 times more number of moves per temperature step

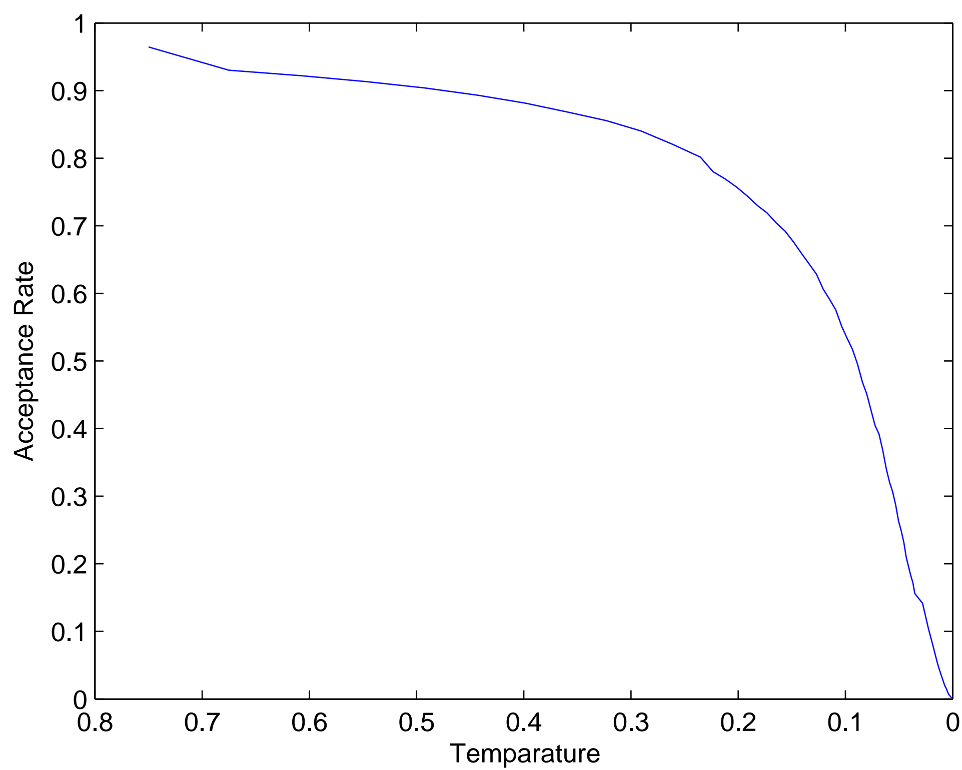

Figure 5.9: Acceptance rate with 64 times more number of moves per temperature step

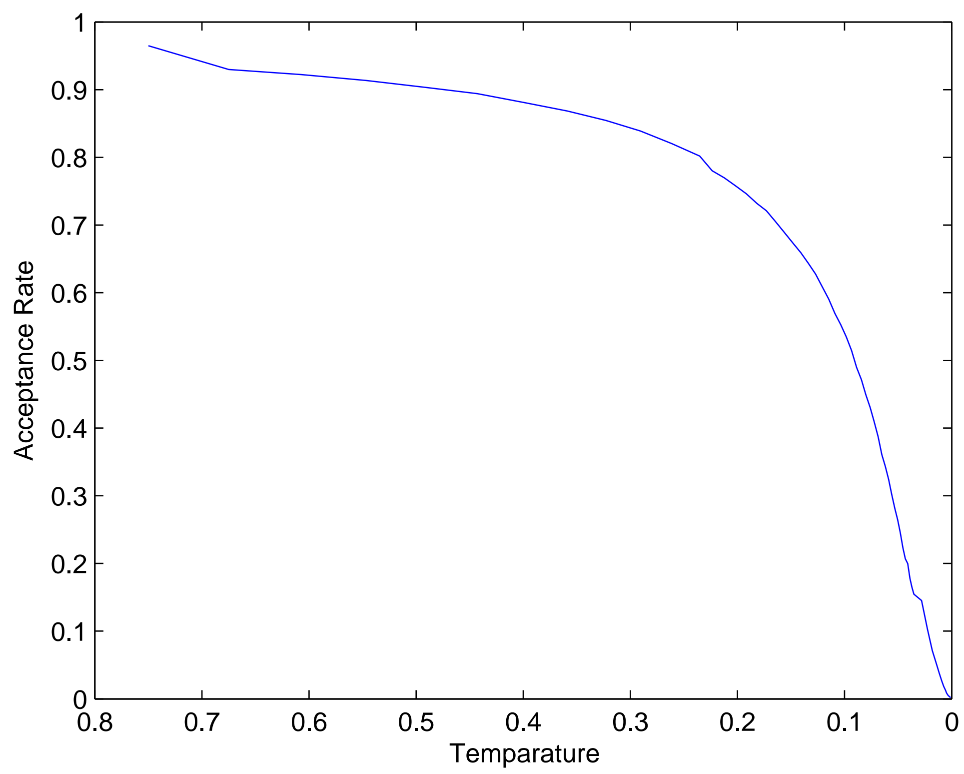

Figure 5.10: Acceptance rate with 256 times more number of moves per temperature step

Chapter 6

Results

The quality of a placement is measured in terms of metrics like the wirelength required for routing the placed circuit, the delay of the circuit etc. The motive for a parallel approach for simulated annealing is to achieve speed over the existing tools. In this chapter we compare both the serial and parallel versions of VPR for speed and quality of placement. We implemented our algorithm on the MCNC benchmarks which are available along with VPR tool. The smallest benchmark among them has 404 logic blocks. Our code worked for benchmarks which had as many as up to 1500 blocks.

6.1 Comparison of Placements

For this analysis we choose the benchmark *e64-4lut* first. The serial VPR takes around 10 seconds of time for placing this circuit. Our algorithm basically divides the task among all the available threads thus reducing the number of iterations per thread. We experimented with different thread configurations and as it can be expected, the speed-up increases with increasing number of threads. Table 6.1 shows the execution time for each thread configuration we tried. For the rest of this document, the unit of time we refer to is a second unless otherwise specified.

For the NVIDIA Tesla GPU we used for this work, 256 is the maximum number of parallel threads that can be run and hence that limits our degree of parallelism. It is clear from the table that for small number of threads the performance of GPUs is worse than the serial version. Also the achieved speed-up is not directly proportional to the

Table 6.1: Comparison of speed-up between different thread configurations

Number of threads	Number of moves by each thread (approximately)	Total time(sec)	Runtime/(Runtime of a single thread on the GPU)
1	30000	758.5	1
2	15000	333.05	0.439
4	7000	151.55	0.199
8	3750	85.16	0.112
16	1800	49.56	0.065
32	900	28.96	0.0381
64	450	15.17	0.02
128	225	8.56	0.011
256	112	5.22	0.0068

number of threads used. This shows that performance benefits on the GPU come with a few overheads which are listed below.

The specifications of the NVIDIA GPU card we used for this analysis are as below

Device 0: "Tesla C1060" Major revision number: 1 Minor revision number: 3 Total amount of global memory: 4294705152 bytes Number of multiprocessors: 30 Number of cores: 240 Total amount of constant memory: 65536 bytes Total amount of shared memory per block: 16384 bytes Total number of registers available per block: 16384 Warp size: 32 Maximum number of threads per block: 512 Maximum sizes of each dimension of a block: 512 x 512 x 64 Maximum sizes of each dimension of a grid: 65535 x 65535 x 1 Maximum memory pitch: 262144 bytes Texture alignment: 256 bytes Clock rate: 1.30 GHz Concurrent copy and execution: Yes

All the benchmarks on the serial code are run using a Pentium 4 - 3.6 GHz processor.

6.2 Analysis of Results

- The memory transfer between the CPU and the GPU
- The synchronization overhead between the threads
- The delay introduced by memory accesses to different levels of GPU memory like global, shared memory etc.

- The difference between the CPU and GPU clock rates(3.6GHz for the CPU Vs 1.3GHz for the GPU)

6.2.1 Synchronization Overhead

The speed up shown in table 6.1 is clearly due to the reduced load on each thread when higher number of threads are engaged. But this is not directly proportional. It is the synchronization overhead that prevents the higher thread configurations from achieving even better speeds. The synchronization between the threads is essential in order to preserve the consistency of the placement. Otherwise, there may be blocks which occupy more than one slot or some blocks might be missing from the circuit. To measure the synchronization overhead, we fix the number of moves performed by each thread-regardless of the number of threads- and measure the time taken by each thread configuration. This comparison is shown in table 6.2

Table 6.2: Comparison of synchronization overhead between different thread configurations

Number of threads	Number of moves by each thread (approximately)	Total time(sec)
1	500	1.88
2	500	2.8
4	500	3.29
8	500	4.11
16	500	9.38
32	500	11.73
64	500	6.42
128	500	6.43
256	500	7.9

As can be expected, the schedule with a single thread consumes the least amount of time. As the number of threads increases, the synchronization overhead increases thus affecting the achieved speed up.

6.2.2 Compute to Memory Access Ratio (CMA)

The factor that most affects the speed up of any application on GPUs is the memory access delay. [31] shows an example of how the *CGMA* (Compute To Global Memory

Access) affects the performance of the GPUs. Though the power of a GPU is as high as it can carry on 327 gigaflops (Floating Point Operations) per second, the global memory bandwidth of NVIDIA G80 GPU is around 80.6 GB/Sec which is just enough to fetch 21.4 billion floating point operators. This dictates the speed-up achieved in the computation.

The same is true in the case of VPR. Most of the operations in the annealing process are additions and comparisons. But these operations frequently need data which has to be fetched from the global memory since that is where the bulk of the data is stored. The data which we transfer from the CPU directly resides in the global memory. Though we have the register memory which is the fastest accessible memory on the GPU, the excess data in the registers also overflows into the global memory. All the major data structures -block, net and clb - which we frequently require to look up are in this memory. This severely limited the speed-up we achieved with so many parallel threads. However, we retain the counter -which we require to keep track of the blocks picked by each thread - in the shared memory.

From the above discussion it is clear that global memory accesses should to be reduced to achieve higher performance. But the constraint is the limited amount of space available in register and shared memories. The register memory is clearly too small for any of the data structures we need. Hence we tried to take advantage of the shared memory. Though the shared memory is insufficient for all the data we need, we store the most critical data structures in the shared memory.

We tried to place the *e64-4lut* circuit with the *block* and *clb* data structures in the shared memory with 4 threads. The placement was done 14% faster with *block* structure and 16% faster with *clb* in the shared memory in comparison with the global memory implementation. The inherent problem of limited space on the GPUs prevents us from going beyond this limited number of threads. Figures 6.1 and 6.2 show the annealing schedule using the shared memory.

A bigger circuit *apex4.net* has also been tried to place with its *clb* structure in the shared memory but did not show any significant change in the speed from that of the previous approach as the global memory implementation can allow far more number of threads.

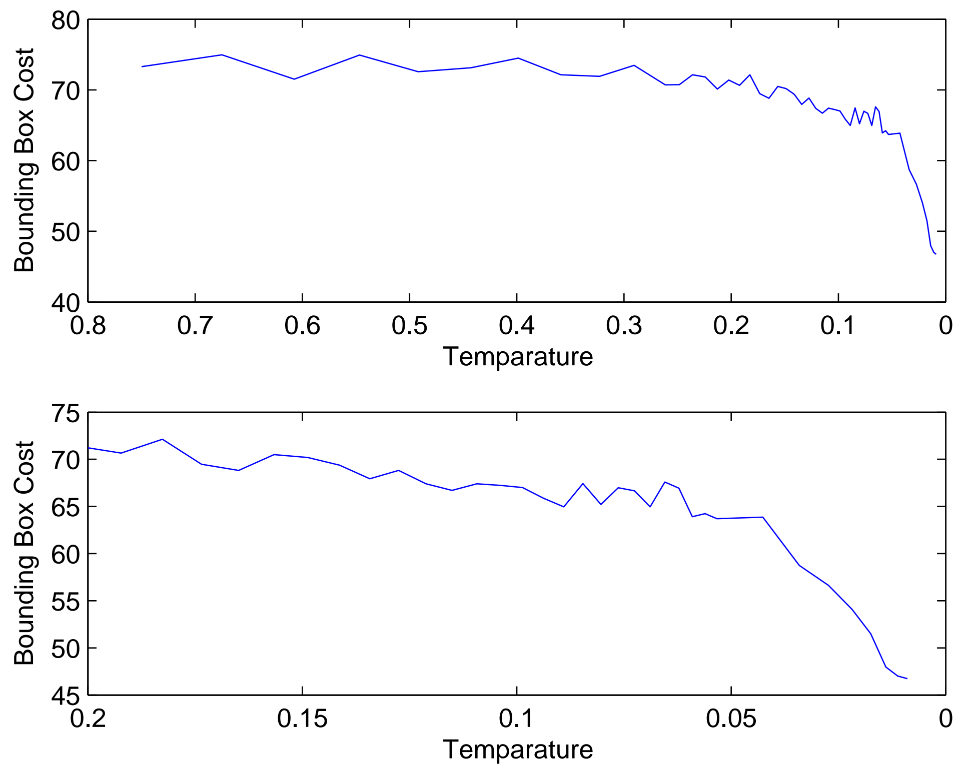

Figure 6.1: Annealing with *block* in the shared memory

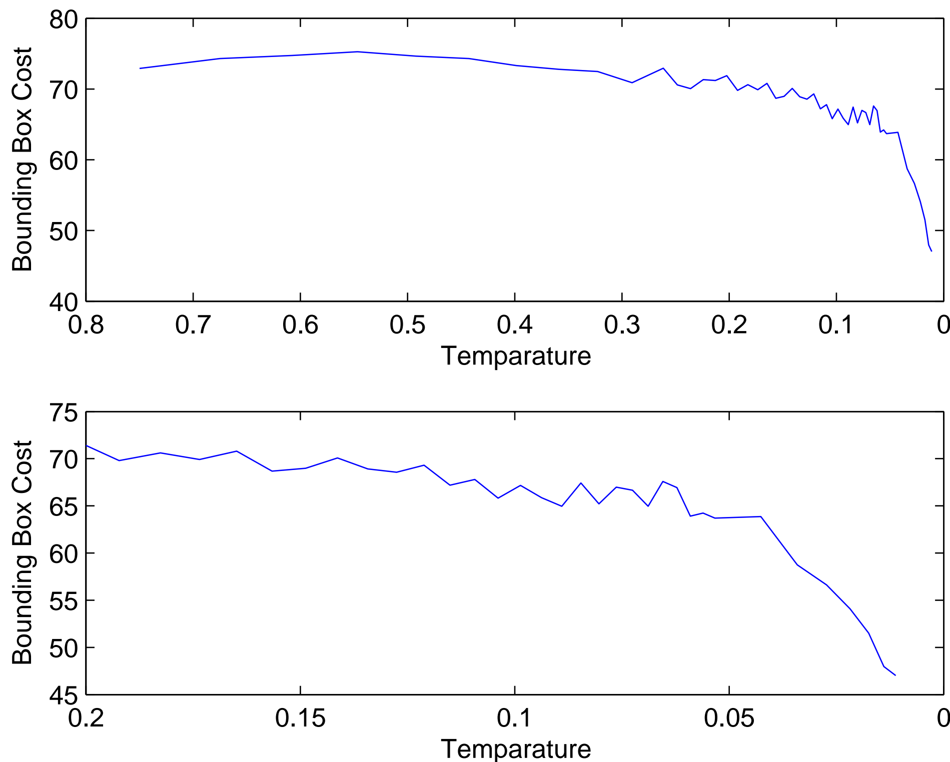

Figure 6.2: Annealing with *clb* in the shared memory

6.3 Comparison of Routing of the Placed Circuits

We now look into the quality-speed tradeoff that results from our parallel implementation. The best way to determine the quality of a placement is to route it. Apart from that, the placement can be judged by the final bounding box achieved and also the time taken to achieve that final value. Figure 6.3 shows the annealing schedule as performed by the original VPR on a CPU. The following figures show how closely our parallel threads method resembles the serial annealing schedule.

In all the figures that follow, the top part shows the annealing total annealing schedule, while the lower portion of the figure gives a closer view of the final stages of the process where around 85% of the optimization takes place.

Figures 6.4- 6.10 show the proximity of our parallel annealing schedule to that of the

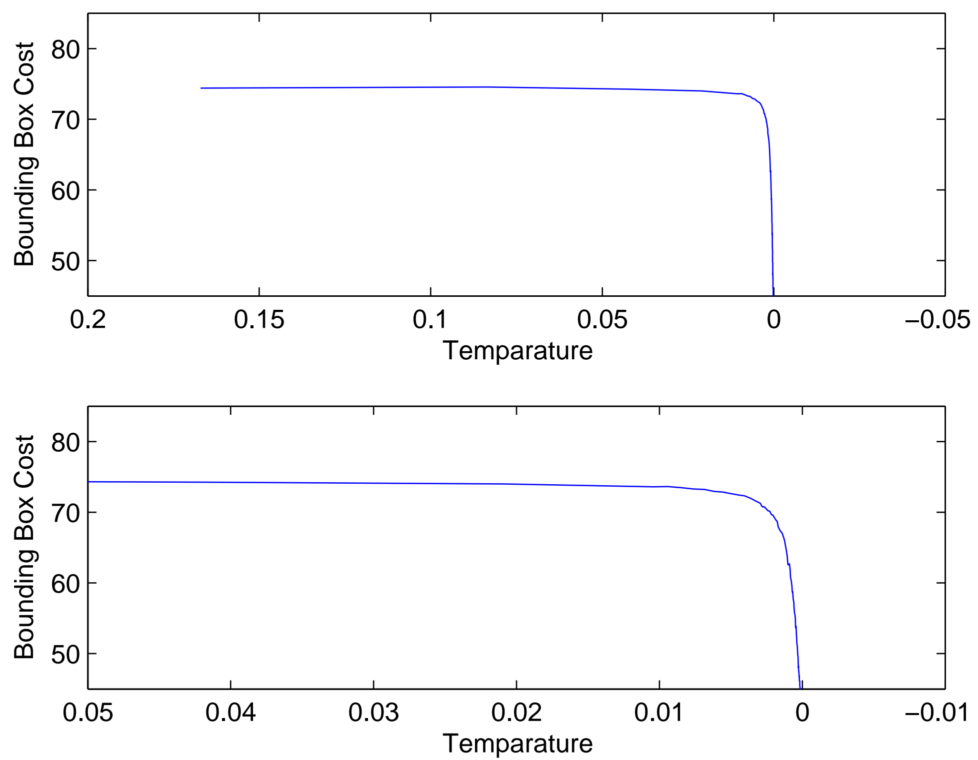

Figure 6.3: The serial annealing schedule

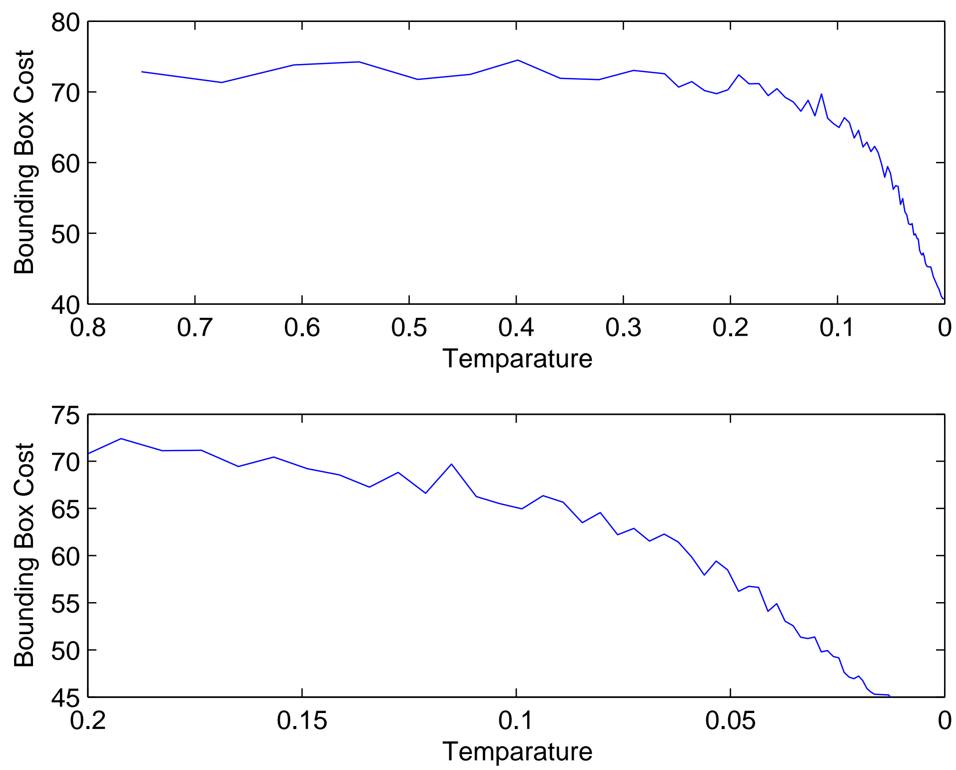

Figure 6.4: Annealing with 4 parallel threads

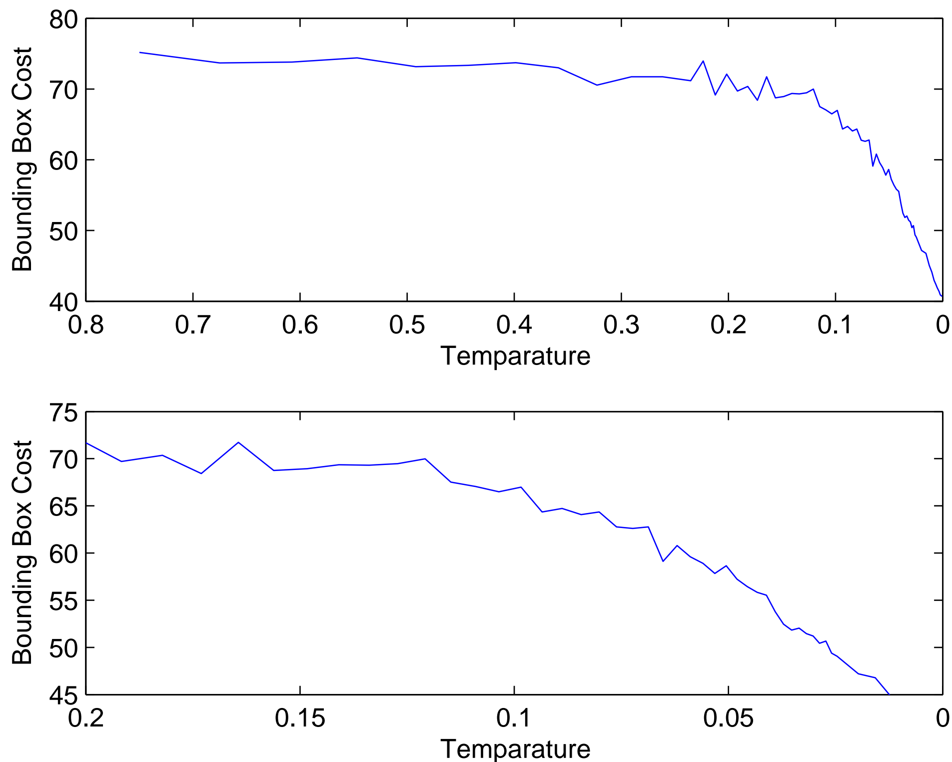

Figure 6.5: Annealing with 8 parallel threads

serial version of VPR shown in figure 6.3 . Though the final bounding box value changes by a small percentage for higher thread configurations, the annealing at large follows a similar schedule. This can be attributed to the frequent synchronization among all the threads. Note that in all the figures, the lower part shows a closer view of the lower temperature stages where about 90% of the optimization takes place. The final bounding box values of different configurations are shown in table 6.3

It should be noted that though the bounding box cost values go on decreasing as in the serial version, our costs do not vary as smoothly. This can be attributed to the non-uniform decrease in our acceptance rates. That is, in the serial version the only factor to accept or reject a move was the temperature. But in our implementation, after the temperature test, we check for block clashes which are random. This makes both our acceptance and cost graphs more roughly varying than those of the original.

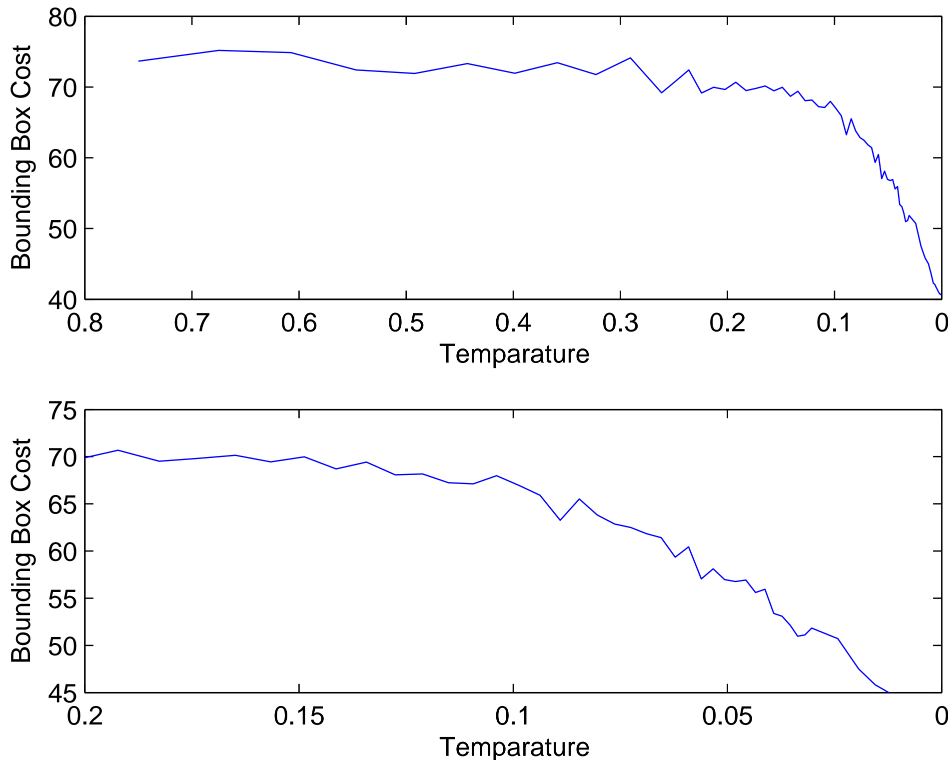

Figure 6.6: Annealing with 16 parallel threads

However, the bounding box cost follows a decreasing trend and is as optimized as the serial annealing algorithm in the final stages.

It should be noted that only at very high thread configurations that the quality of our placement is degraded by a maximum of 26% when 70% speed-up is achieved. At lower speeds the same quality of placement has been maintained. The degradation of the quality can be attributed to the intermediate temperature steps that have been missed. The higher number of temperature steps visited by the serial version allows that to explore more solution space and hence gives a better quality.

The quality of a placed design can be determined by routing the design. We routed our placement files using the VPR's serial router and compared various metrics of routing. Table 6.4 summarizes the results of the quality tests.

The maximum and minimum percentage change in metrics is shown in table 6.5

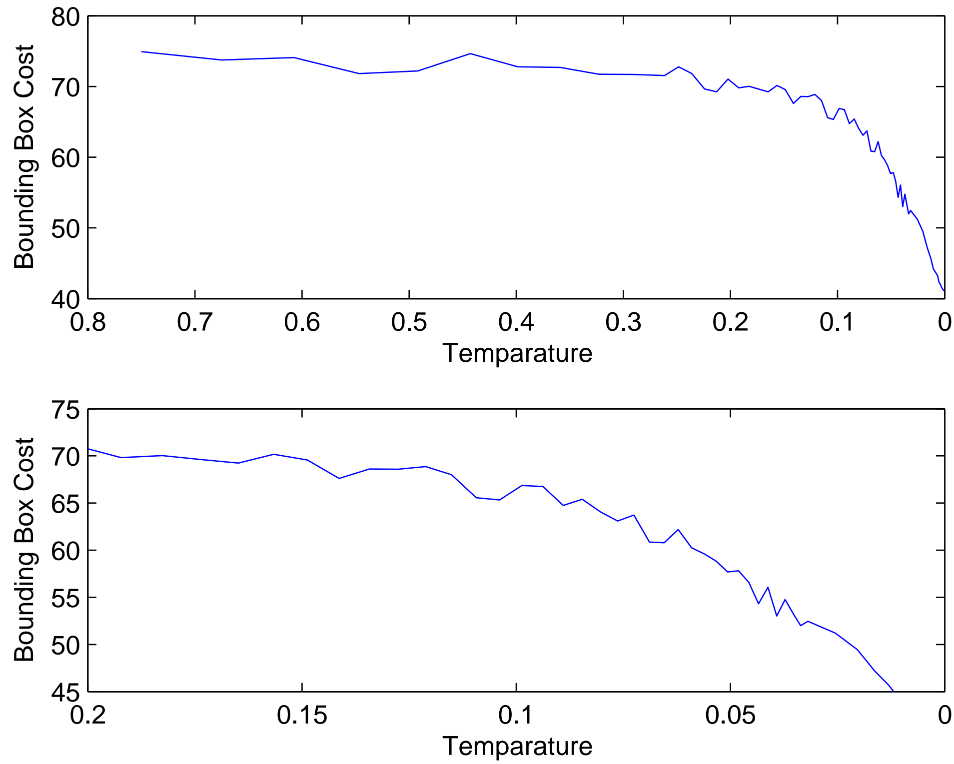

Figure 6.7: Annealing with 32 parallel threads

Table 6.3: Comparison of final bounding box values between different thread configurations

Number of threads	Final Bounding Box achieved	Time taken
serial	42.23	8.53
4	43.1	151.55
8	44.6	85.16
16	44.4	49.56
32	44.5	28.96
64	45.0	15.17
128	47.3	8.56
256	53.1	5.22

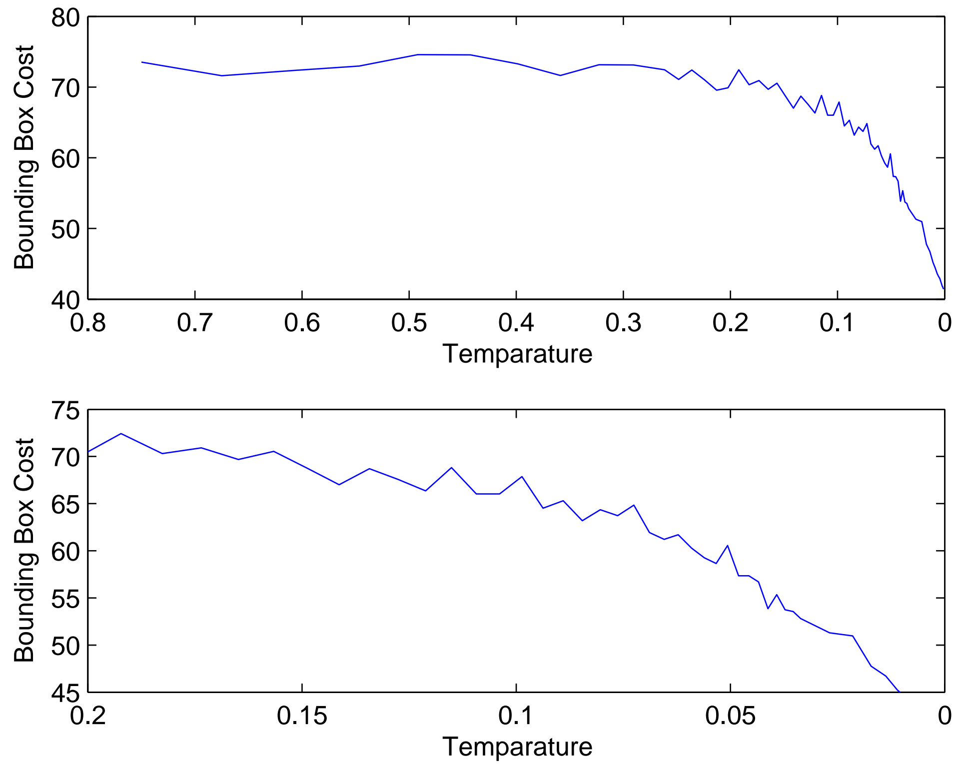

Figure 6.8: Annealing with 64 parallel threads

Table 6.4: Comparison of quality metrics between different thread configurations

Number of Threads	Channel Width Factor	Maximum Number of bends	Total Wirelength	Maximum Netlength	Total Logical Delay	Total Net Delay	Critical Path
Serial	12	39	4915	92	2.9ns	44ns	47ns
4	12	40	5036	107	2.9ns	51ns	54ns
8	12	33	5002	86	2.4ns	51ns	54ns
16	12	38	5245	100	2.4ns	53ns	55ns
32	12	34	5029	84	2.9ns	53ns	56ns
64	12	46	5231	103	2.9ns	55ns	57ns
128	13	40	5397	105	2.9ns	54ns	57ns
256	15	35	6255	95	2.9ns	65ns	68ns

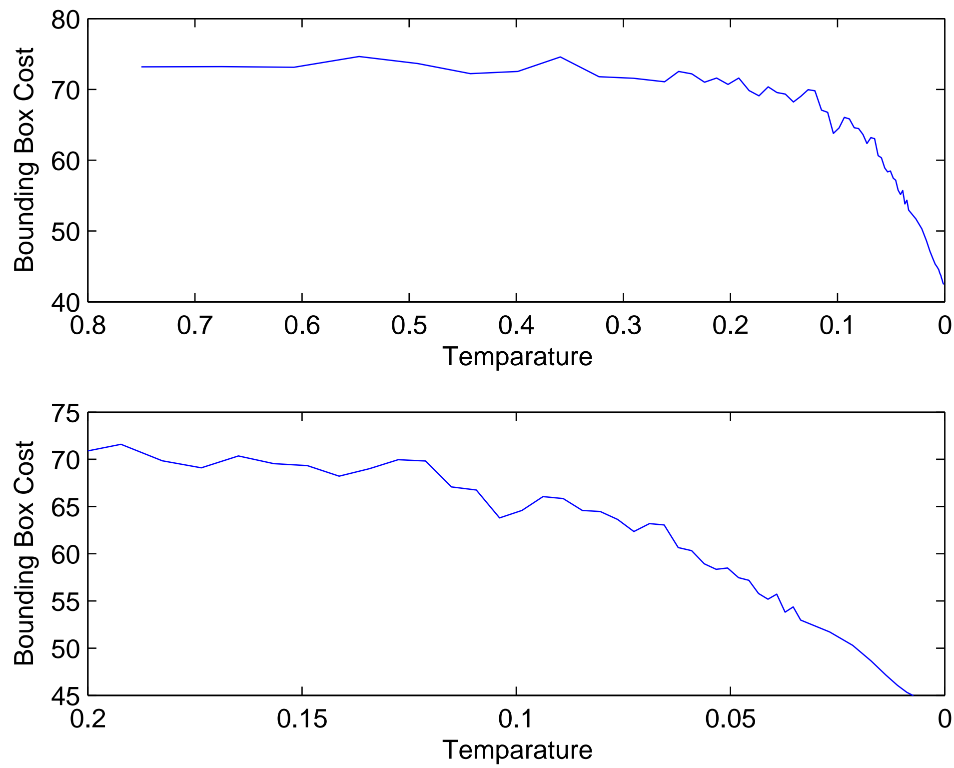

Figure 6.9: Annealing with 128 parallel threads

Table 6.5: Degradation in quality metrics by the parallel method

Metric	Maximum % Deviation
Channel Width Factor	25
Max. Number of Bends	17
Total Wirelength	26.6
Maximum Netlength	16.3
Total Logical Delay	17
Total Net Delay	47.7
Critical Path	44.6

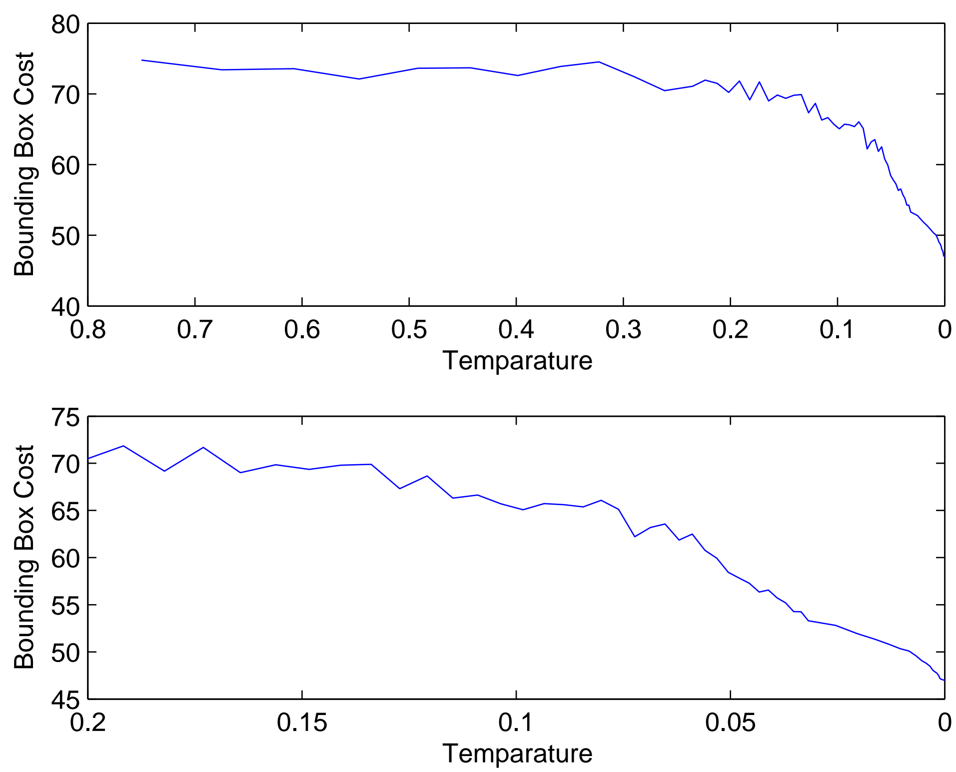

Figure 6.10: Annealing with 256 parallel threads

Chapter 7

Conclusion and Future Work

A parallel multi threaded approach for simulated annealing has been proposed. We implemented our algorithm on the Versatile Place and Route(VPR) placer. The main challenges involved in parallelizing simulated annealing are the inherent data dependency between different stages, the communication required between the different parallel branches to maintain the accuracy of the original algorithm.

In this work we analyzed various approaches to parallelize the placement problem and implemented the approach most suitable for the GPUs. Though dividing the various stages of placement among different threads has been widely discussed in literature, keeping in view the divergence problem of GPUs, we adopted the method where each thread implements the whole algorithm but for a reduced number of times thus reducing the total execution time of the program.

Various methods of inter-thread communications have been proposed and contrasted. Keeping in view the limited on-chip memory available on the GPUs, we resorted to a simple counter based method to communicate between different threads in order to maintain the placement consistency.

The most significant speed bottleneck in GPU implementations arises from memory access delays. As with any processor, the faster the memory the lower the size. Different ways to efficiently utilize the available limited memory have been shown. All levels of memory including shared, register and global memory have been taken advantage of by the implementation.

Our method has performed 37% faster than the VPR for a benchmark as big as 400

logic blocks with only 17% degradation in the wirelength. The method performed at almost equal speed to VPR for benchmarks that had more than 1000 logic blocks. The problem with this implementation was that the higher the degree of parallelism, the higher the speed achieved. On the other hand, the limited memory on GPUs does not suffice for more number of threads for bigger circuits.

The limited on-chip memory available on GPUs, higher access times for the global memory are major drawbacks for GPUs. Applications like VPR where the data to be handled is of large size, are severely limited by these problems.

[31] has shown that the maximum advantage of GPUs can be taken when the arithmetic operations to memory accesses ratio is increased. In other words, in arithmetic-intensive applications where data once fetched is used for many calculations, GPUs are the best suited. The problem of placement is more data intensive where data reads and writes are much higher than the cost calculations and comparisons. This posed a major challenge to our implementation limiting our speed-up to only 37% when compared to the serial version.

Potential improvements can be made to this proposed method which would enhance the performance. The acceptance rates of our annealing schedule deviate much from that of the serial version. When handling the block clashes, we discard both the moves which have a clash between them. Coming up with a method to prioritize and choose between conflicting threads might help closely resemble the serial annealing schedule.

Partitioning the data and efficiently scheduling it to faster memories (shared and register memories), can further take the advantage of the memory hierarchy for achieving higher speeds.

References

[1] Vaughn Betz, Jonathan Rose, and Alexander Marquardt. Architecture and cad for deep-submicron fpgas. *Kluwer Academic Publishers*, 1999.

[2] Stephen D Brown, Robert J Francis, Jonathan Rose, and Zvonko G Vranesic. Field-programmable gate arrays. *Kluwer Academic Publishers*, 1992.

[3] J.He and J.Rose. Advantages of heterogenous architectures for fpgas. *CICC*, pages 7.4.1–7.4.5, 1993.

[4] K.Chung, S.Singh, J.Rose, and P.Chow. Using hierarchical logic blocks to improve the speed of field programmable gate arrays. *Int Workshop on Field Programmable Logic and Applications*, 1991.

[5] Xilinx Inc. The programmable logic data book. 1994.

[6] Altera Inc. Data book. 1998.

[7] Lucent Technologies. Fpga data book. 1998.

[8] Actel Inc. Fpga data book and design guide. 1994.

[9] Xilinx Inc. Ise in-depth tutorial. 2010.

[10] A.Dunlop and B.Kernighan. A procedure for placement of standard cell vlsi circuits. *IEEE Trans. on CAD,*, pages 92–98, 1985.

[11] Pongstorn Maidee, Cristinel Ababei, and Kia Bazargan. Timing-driven partitioning-based placement for island style fpgas. *IEEE transactions on Computer-Aided Design for Integrated Circuits and Systems*, 2005.

[12] J.Kleinhans, G.Sigl, F.Johannes, and K.Antreich. Gordian: Vlsi placement by quadratic programming and slicing optimization. *IEEE Trans. on CAD*, pages 356–365, 1991.

[13] C.Alpert, T.Chan, D.Huang, P.Mulet A.Kahng, I.Markov, and K.yan. Faster minimization of linear wirelength for global placement. *ACM Symp. on Physical Design*, pages 4–11, 1997.

[14] G.Sigl, K.Doll, and F.Johannes. Analytical placement: A linear or a quadratic objective function? *DAC*, pages 427–432, 1991.

[15] John A. Chandy and Prithviraj Banerjee. Parallel simulated annealing strategies for vlsi cell placement. *International Conference on VLSI Design*, 1996.

[16] Ellen E.Witte, Roger D.Chamberlain, and Mark A.Franklin. Parallel simulated annealing using speculative computation. *IEEE Transactions on Parallel and Distributed Systems*, 2, 1991.

[17] V.Betz and J.Rose. Vpr:a new packing, placement and routing tool for fpga research. *International Workshop on Field-Programmable Logic and Applications*, pages 213–222, 1997.

[18] F.Romeo M.Huang and A.Sangiovanni-Vincentelli. An efficient general cooling schedule for simulated annealing. *ICCAD*, pages 381–384, 1986.

[19] W.Swartz and C.Sechen. New algorithms for the placement and routing of macro cells. *ICCAD*, pages 336–339, 1990.

[20] Jonas Knopman and J'ulio S Aude. Parallel simulated annealing:an adaptive approach. *11th International Parallel Processing Symposium*, 1997.

[21] Maylay Haldar, Anshuman Nayak, Alok Choudary, and Prith Benerjee. Parallel algorithms for fpga placement. *10th Great Lakes Symposium on VLSI*, 2000.

[22] John A. Chandy, Sungho Kim, Balakrishna Ram kumar, Steven Parkes, and Prithviraj Banerjee. An evaluation of parallel simulated annealing strategies with application to standard cell placement. *IEEE transactions on Integrated Circuits and Systems*, 1997.

[23] Michael Wrighton and Andr'e M.DeHon. Hardware-assisted simulated annealing with application for fast fpga placement. *11th International Symposium on Field Programmable Gate Arrays*, 2003.

[24] Soo-Young Lee and Kyung Geun Lee. Synchronous and asynchronous parallel simulated annealing with multiple markov chains. *IEEE Transactions on Parallel and Distributed Systems*, 7, 1996.

[25] Milena Lazarova. Parallel simulated annealing for solving the room assignment problem for shared and distributed memory platforms. *9th International Conference on Computer Systems and Technologies*, 2008.

[26] King-Wai Chu, Yuefan Deng, and John Reinitz. Parallel simulated annealing by mixing of states. *Journal of Computational Physics*, 148, 1999.

[27] Daniel R.Greening and Frederica Darema. Rectangular spatial decomposition methods for parallel simulated annealing. *Physica D: Nonlinear Phenomena*, 42, 1989.

[28] R.D. Chamberlain, M.N. Edelman, M.A. Franklin, and E.E Witte. Timing-driven partitioning-based placement for island style fpgas. *Computer Design: VLSI in Computers and Processors*, 1988.

[29] K.L. Wong and A.G. Constantinides. Speculative parallel simulated annealing with acceptance prediction. *IEEE transactions on Computers and Digital Techniques*, 143:219–223, 1996.

[30] C.S. Jeong and M.H. Kim. Fast parallel simulated annealing for travelling salesman problem. *Neural Networks*, 1990.

[31] David B Kirk and Wen mei W Hwu. Programming massively parallel processors: A hands on approach. 2008.