CHAPTER 3

PARALLEL MEMETIC ALGORITHM FOR VLSI CIRCUIT PARTITIONING PROBLEM USING GPGPU

3.1 INTRODUCTION

With rapid advancements in integration technology, it is possible to place a large number of logic gates on a single chip. Despite this fact, it may become necessary to partition a circuit into several sub-circuits and implement the sub-circuits as ICs. Because, either the circuit is too large to be placed on a single chip, or because of I/O pin limitations. The larger the gate count of the circuit, the larger is the number of I/O pins associated with the circuit. Unfortunately, a large pin count increases the cost of packaging the circuit dramatically. When it becomes necessary to split a circuit across packages, care must be taken as to how this partition is carried out.

Invariably some interconnections will get ‘cut’ when a circuit is partitioned into sub-circuits, while interconnections within the sub-circuits can be implemented as ‘on-chip’ wiring. Off-chip wires are undesirable due to several reasons. Electrical signals travel slower along wires external to the chip; thus off-chip wires cause performance degradation. Off-chip wiring reduces the reliability of the system. Finally, since off-chip wires must originate and terminate into I/O pins, more off-chip wires mean more I/O pins.

Unfortunately, the partitioning problem is NP-complete problem; this means it is unlikely that a polynomial-time algorithm exists for solving
the problem. Therefore, one must use heuristic techniques generating approximate solutions. Partitioning is one of the first steps in VLSI circuit design. Partitioning has the important responsibility since it directly affects the rest of the steps in the process. A bad partitioning algorithm could leave a very well area-balanced chip, but with terrible wiring. It could also be ended-up with a partition, which allows least complex wiring, but with partitions being uneven in size.

These results are undesirable, thus various partitioning algorithms have been built to create good partitions. Universally, all partitioning algorithms are expected to give good partitions, which are defined as maintaining area constraints and minimizing wiring complexity. In fact, the perfect partition could always be found with any partitioning problem given an infinite amount of time. Time is limited; however, the algorithms must arrive at a reasonably good solution in polynomial time.

The circuit-partitioning problem consists of finding a decomposition of the target circuit into non-overlapping sub-circuits with at least one logical gate in each sub-circuit. The different objectives that may be satisfied by the desired partitioning are (i) the minimization of the number of cuts, (ii) the minimization of the deviation in the number of elements (inputs, logical gates, outputs and fan-out points) assigned to each partition.

The circuit-partitioning problem can be taken as a graph partitioning problem, where each module (gates etc.) is taken as vertex and the connection between them representing the edges between the nodes. It is very difficult to solve with conventional techniques such as exact or approximations algorithms. Because of its complexity, many heuristic algorithms have been developed to solve it with varying degrees of success.
In this chapter, a heuristic algorithm is presented to solve the graph partitioning problem. The algorithm incorporates several MA features as well as local optimization techniques and graph preprocessing. The algorithm produced very good results and in some cases equaled the best known results. In section 1, an introduction to the graph partitioning problem is presented. Section 2 includes a definition of the problem and problem formulation. Section 3 describes how the MA concept can be applied to graph partitioning. The detail of the partitioning using Memetic Algorithm comprises Genetic Algorithm as a global search and Simulated Annealing as a local search are discussed. Section 4 gives an overview of parallel computation on GPU. Section 5 describes about parallel Memetic Algorithm framework for solving partitioning problem. A number of experiments have been performed and the experimental results are discussed in Section 6. Conclusions are given in the last section.

3.2 PROBLEM DEFINITION

This work addresses the problem of VLSI circuit partitioning (netlist partitioning) with the objective of optimizing cutset while considering the Balance constraint (same as area constraint as unit area is assumed for every gate). Formally, the problem can be stated as follows: Given a set of modules \( V = \{v_1, v_2, \ldots, v_n\} \), the purpose of partitioning is to assign the modules to a specified number of clusters \( k \) (two in our case) satisfying prescribed properties. In general, a circuit can have multi-pin connections (nets) apart from two-pin and therefore it is better to represent it by a hypergraph.

A hypergraph \( H(V,E) \) is defined where \( V \) is a set of nodes and \( E \) is a set of hyperedges. Node \( v_i \in V \) corresponds to an element (e.g., a gate) in the circuit, and hyperedge \( e_i \in E \) corresponds to a net in the circuit. Given a hypergraph \( H(V,E) \) with \( E = \{e_1, e_2, \ldots, e_m\} \) being the set of signal nets, each net is a subset of \( V \) containing the modules connecting the net. It is assumed
that for each hyperedge \( e \in E, |e| \geq 2 \) (it connects at least two nodes). Our task is to divide \( V \) into 2 subsets (clusters) \( V_0 \) and \( V_I \) in such a way that the objectives are optimized, subject to balance constraints.

**Cutsize:** The set of hyperedges cut by a cluster \( C \) is given by \( E(C) = \{ e \in E: 0 < |e \cap C| < |e| \} \) i.e., \( e \in E(C) \) if at least one, but not all, of the pins of \( e \) are in \( C \). The set of nets cut by a partitioning solution \( p^k \) can be expressed as

\[
E(p^k) = \bigcup_{i=1} E(c_i) \tag{3.1}
\]
or equivalently \( E(p^k) = \{ e \in E | \exists u, v \in e; h \neq l \text{ with } u \in C_h \text{ and } v \in C_l \} \). We say that \( |E(p^k)| \) is the cutsize of \( p^k \). The cost function \( f \) can be written as follows:

\[
f = \sum_{e \in \psi} w(e) \tag{3.2}
\]

where \( \psi \in E \) denotes the set of off-chip edges, i.e., nets cut. The weight \( w(e) \) on the edge \( e \) represents the cost of wiring the corresponding connection as an external wire. If all weights equal one, the cost function becomes simpler:

\[
f = |\psi| \tag{3.3}
\]

where \( |\psi| \) denotes the cardinality of the set \( \psi \).

**Area or Balance Constraint:** If we assume that the area of all cells is identical, then the problem reduces to balancing the two partitions in terms of the number of cells. The balance constraint is given below:

\[
\frac{|\beta_h - \beta_l|}{\psi} \leq \alpha \tag{3.3}
\]
where $\beta_i$ is the number of cells in partition $i$, $\phi$ is the total number of cells in the circuit, $\alpha$ is the tolerance which is equal zero in case of a perfect balance. When balance is used as cost, it will be $|\beta_1 - \beta_2|$.

3.3 MEMETIC ALGORITHM FOR PARTITIONING PROBLEM

The pure Genetic Algorithms are not well suited to fine tuning search in complex combinatorial spaces and that hybridisation with other techniques can greatly improve the efficiency of search. Memetic Algorithm (MA) is a form of population-based hybrid genetic algorithm (GA) coupled with an individual learning procedure capable of performing local refinements. Here genetic algorithm is used as global search to explore the search space and simulated annealing as a local search method to exploit the information in the search region for the optimization of VLSI netlist bi-partitioning problem.

3.3.1 Genetic Algorithm

GA is an elegant search technique that emulates the process of natural evolution as a means of progressing towards the optimal solution. GAs start from a population of solutions and creates new generations by means of crossover and mutations, the good characteristics of selected good parents are supposed to be preserved and the survival of the fittest should guarantee the improvement of the solution.

Chromosome Encoding and Initial Solution

Solution representation has a major role in designing an efficient GA. This representation should permit a large diversity in a small population. Here conventional encoding scheme was preferred, where bi-partition are
represented by strings “0” and “1”. “0” cells represent that the modules reside in partition group 1 and vice versa as shown in Figure 3.1.

![Figure 3.1 Chromosome Representation for Circuit Partitioning](image)

**Initial Population Generation**

The algorithm starts with a set of initial solutions called population that is generated randomly or taken from the results of a constructive algorithm. When generating the random initial solution it is preferred that it is within the bounds of the balance constraint. This population is obtained by generating random covers of compatible subsets of nodes. A random choice of size of each subset in the cover contributes to some diversity in the population. The diversity of the population of solution is verified experimentally and the method of initial population generation is also verified for adequate diversity.

**Genetic Crossover Operators**

The first genetic operation done to the chromosomes in the mating pool is crossover. The idea behind crossover is to create an information exchange between two chromosomes. By doing so, the algorithm will explore
new paths and hopefully be able to find better paths in the process. Here Partially Mapped Crossover (PMX) is used to create offspring from the parents. In PMX, as the repetition of cells is avoided by a mapping function, unbalance group formation is completely eliminated. Therefore, PMX finds many new offspring's without increasing computational complexity.

Partially Mapped Crossover (PMX) is a crossover of permutations which guarantees all positions that will be found exactly once in each offspring, i.e. both offspring receive a full complement of genes, followed by the corresponding filling in of alleles from their parents. Basically, parent 1 donates a swath genetic material and the corresponding swath from the other parent is sprinkled about in the child. Once that is done, the remaining alleles are copied direct from parent 2. PMX proceeds as follows:

1. Randomly select a swath of alleles from parent 1 and copy them directly to the child. Note the indexes of the segment.

2. Looking in the same segment positions in parent 2, select each value that "hasn't already been copied" to the child.

A. For each of these values:

i. Note the index of this value in Parent 2. Locate the value, \( V \), from parent 1 in this same position.

ii. Locate this same value in parent 2.

iii. If the index of this value in Parent 2 is part of the original swath, go to step i using this value.

iv. If the position isn't part of the original swath, insert step A's value into the child in this position.

3. Copy any remaining positions from parent 2 to the child.
**PMX Example**

*Parent 1*: 8 4 7 3 6 2 5 1 9 0  

*Parent 2*: 0 1 2 3 4 5 6 7 8 9  

*Child 1*: __ __ 3 6 2 5 1 __

1. A random swath of consecutive alleles from Parent 1 to the child is copied.

*Parent 1*: 8 4 7 3 6 2 5 1 9 0  

*Parent 2*: 0 1 2 3 4 5 6 7 8 9  

*Child 1*: __ __ 3 6 2 5 1 __

2. ‘4’ is the first value in the swath of Parent 2 that isn’t in the child. ‘6’ is identified as the value in the same position in Parent 1. The value ‘6’ is located in Parent 2 and noticed that it is still in the swath. So, using ‘6’ as the value go back to step ‘i’.

*Parent 1*: 8 4 7 3 6 2 5 1 9 0  

*Parent 2*: 0 1 2 3 4 5 6 7 8 9  

*Child 1*: __ __ 3 6 2 5 1 __

3. Repeating Step i: Once again, it can be seen that ‘5’ is in the same position in Parent 1, and locate ‘5’ in Parent 2. It is also in the swath, so repeat step ‘i’ once more with ‘5’ as value.
Parent 1 : 8 4 7 3 6 2 5 1 9 0

Parent 2 : 0 1 2 3 4 5 6 7 8 9

Child 1 : _ _ 4 3 6 2 5 1 _ _

4. Repeating Step i: It can be seen that ‘2’ is in the same position in Parent 1, and it is located in Parent 2 in the 3rd position. Finally, position in the Child for the value ‘4’ from Step 2 is obtained.

Parent 1 : 8 4 7 3 6 2 5 1 9 0

Parent 2 : 0 1 2 3 4 5 6 7 8 9

Child 1 : _ 7 4 3 6 2 5 1 _ _

5. ‘7’ is the next value in the swath in Parent 2 that isn't already included in the Child. So, the same index in Parent 1 is checked and a ‘1’ is seen in that position. Next, ‘1’ is checked in Parent 2 and found in the 2nd position. Since the 2nd position is not part of the swath, a home for the value ‘7’ is found.

Parent 1 : 8 4 7 3 6 2 5 1 9 0

Parent 2 : 0 4 2 3 4 5 6 7 8 9

Child 1 : 0 7 4 3 6 2 5 1 8 9

6. Now the easy part, all swath values have been taken care of, so everything else from Parent 2 drops down to the child.
Mutation operator

A mutation operator is usually used to ensure that an optimal solution can be found with a probability greater than zero. This operator could improve the performance of an EA, given its ability to continue the search of global optimal (or near optimal) solutions even after local optimal solutions have been found, not allowing the algorithm to be easily trapped in local sub-optimal solutions. Each time that an offspring chromosome is generated, a mutation probability, $P_m$ is used to decide whether the mutation operator should be applied to this chromosome. In bi-partitioning problems considered in this thesis, depending on the mutation rate, a few nodes are selected randomly from the chromosome, and replaced in the other possible nodes in the chromosome.

Next Generation Population Selection

Selection plays an important role in improving the average quality of the population by passing the high quality chromosomes to the next generation. There are several techniques by which a mating pool for reproduction can be created. Such a mating pool is characterized by single or multiple copies of the fittest parent chromosomes. The probability of selection and the expected number of copies are measures that are used to decide the number of copies each individual gets into the mating pool. Here individuals for the next population are selected based on the elitist-random selection (ernd). $N_p/2$ ($N_p$ is the population size) best chromosomes are selected and the remaining $N_p/2$ are selected randomly. Based on experimental results this scheme offers better choice than other schemes, because it provides balance between greediness and randomness.
3.3.2 Local Search

In many optimization problems, it is possible to get caught in a local minimum. In other words, the algorithm will produce a “minimal” solution that is not even close to minimal, but be unable to find the true minimum, because the only way to get there is to temporarily create a solution that is worse than the one the algorithm started with. Computer algorithms are notoriously bad at doing “bad” things for the purpose of “getting something even better.” One algorithm that was specifically designed to get around this problem is Simulated Annealing.

Simulated Annealing (SA) is a general iterative improvement algorithm that can be used for many different purposes. In SA, it is necessary to consider several thousand or even several million states, so computing a new state from an old state must be very efficient. In partitioning, SA starts with a random partition, just as the two previous algorithms. A new state is computed by selecting a gate at random from each of the two subsets, and swapping them. As given before, the swap remains tentative, until the quality of the new partitioning is computed. The number of nets cut is the measure of goodness. If the new state is better than the old state, it is accepted and the swap is made permanent. If the new state is worse than the old state, it might be accepted and it might not.

The SA algorithm operates in a series of distinct phases called “temperatures.” An actual temperature value is assigned to each phase. The algorithm begins with temperature set to a high value, and proceeds to lower and lower temperatures. A predetermined number of moves is attempted at each temperature. When a bad move is attempted, the algorithm computes an acceptance value that is based on temperature and on the “badness” of the
solution. This acceptance value is compared to a random number to determine whether the move will be accepted. The random number is used to guarantee that there is always a nonzero probability that any bad move will be accepted. The higher the temperature, the more likely is a particular bad move will be accepted, and at a given temperature, the worse the move, the less likely it is to be accepted.

In most cases the acceptance function is computed using the following function, \( e^{\frac{-\delta_s}{T}} \) where \( \delta_s \) is the change in the quality and \( T \) is the current temperature. For bad moves this function will produce a value between 0 and 1. A random number between 0 and 1 is generated and if the quality measure is larger than the generated random number, the bad move is accepted. Recall that in partitioning, negative values of \( \delta_s \) are good and positive values are bad. There are several parameters that must be determined experimentally. The first among these is the starting temperature. This is usually chosen so that many bad moves will be accepted. The second is the cooling schedule. This is the series of temperatures that will be used. The change in temperature is seldom uniform. Usually temperature is changed in large steps at high temperatures and in small steps at lower temperatures. A significant amount of experimentation is usually necessary to determine the best cooling schedule. The final parameter that must be determined is the number of moves to be made at each temperature. This number of moves is generally based both on temperature and on the number of gates in the input. One reasonable choice for a number of moves is 500 times the number of gates in the input. Simulated annealing generally does a very good job, but runs very, very slowly. An outline of the algorithm SA is as follows:
procedure SA;
begin
  t = t0;
  cur_part = ini_part;
  cur_score = SCORE (cur_part);
  repeat
    repeat
      compl = SELECT (part1);
      comp2 = SELECT (part2);
      trial_part = EXCHANGE (compl, comp2, cur_part);
      trial_score = SCORE (trial_part);
      δs = trial_score – cur_score;
      if (δs < 0) then
        cur_score = trial_score;
        cur_part = MOVE (compl, comp2);
      else
        r = RAND (0,1);
        if (r < exp(- δs/t)) then
          cur_score = trial_score;
          cur_part = MOVE (compl, comp2);
      until (equilibrium at t is reached);
  t = αt; /* 0 < α < 1 */
  until (freezing point is reached);
end

Figure 3.2 Outline of Simulated Annealing
3.3.3 Memetic Algorithm (MA)

Initially, MA randomly generates a population of individuals using the technique described above. Then, MA starts evolving the population generation by generation. In each generation, MA uses the genetic operators probabilistically on the individuals in the population to create new promising search points (admissible partitioning) and uses the local search method to optimize them. The process is repeated until a preset generation is up. An outline of MA is as follows:

Memetic Algorithm for partitioning problem

generate initial solutions(random)

while(stop criteria is not met)

crossover(PMX)
mutation(rand exchange module)
local search(SA)
calculate fitness(cut size)
selection(elitist-random selection)

end while

End

Figure 3.3 Pseudocode of a Memetic Algorithm

3.4. GRAPHICS PROCESSING UNIT

Three major factors make the development of graphics hardware based on commodity PCs truly outstanding in recent years. First, the computational power of Graphics Processing Units (GPUs) for commodity PC hardware has grown much faster than for CPUs. Second, the high performance is available at a very good cost/performance ratio. Finally, within the last 4-5 years, GPUs have become programmable by high level
languages. From an abstract point of view, the GPU is a parallel streaming processor, particularly suitable for the fast processing of large arrays. Thus, many researchers have started utilizing graphics processors to enhance the performance of their specific, in many cases, non-graphics applications and simulations. The special field of “General-Purpose computation on GPU (GPGPU)” has evolved offers a survey of this emerging research area. Although performance gains depend strongly on the application, one can say that speedup factors around 5 against algorithms on the CPU are commonly reported.

Graphics Processing Units (GPUs) are fast, highly parallel processor units. In addition to processing 3D graphics, modern GPUs can be programmed for more general purpose computation. The GPU consists of a large number of ‘shader processors’, and conceptually operates as a Single Instruction Multiple Data (SIMD) or Multiple Instruction Multiple Data (MIMD) stream processor. A modern GPU can have several hundred of these stream processors, which combined with their relatively low cost, makes them an attractive platform for scientific computing.

Graphics processors are specialized stream processors used to render graphics. Typically, GPU is able to perform graphics manipulations much faster than a general purpose CPU, as the processor is specifically designed to handle certain primitive operations. Internally, the GPU contains a number of small processors that are used to perform calculations on 3D vertex information and on textures. These processors operate in parallel with each other, and work on different parts of the problem. First the vertex processors calculate the 3D view, and then the shader processors paint this model before it is displayed. Programming the GPU is typically done through a virtual machine interface such as OpenGL or DirectX which provide a common interface to the diverse GPUs available thus making development
easy. However, DirectX and OpenGL are optimized for graphics processing, hence other APIs are required to use the GPU as a general purpose device. Depending on the GPU, the number of instructions may be limited. In order to use more than this number of operations, a program needs to be broken down into suitably sized units, which may impact performance. Newer GPUs support unlimited instructions, but some older cards support as few as 64 instructions. GPUs typically use floating point arithmetic, the precision of which is often controllable as less precise representations are faster to compute with. Again, the maximum precision is manufacturer specific, but recent cards provide up to 128-bit precision.

The rapid increase in the number and diversity of scientific communities exploring the computational power of GPUs for their data intensive algorithms has had a key contribution in encouraging GPU manufacturers to design more powerful, easily programmable and flexible GPUs. In addition, the development of open-source programming tools and languages for interfacing with the GPU platforms has further fueled the growth of general purpose GPU (GPGPU) applications. Further, GPU architectures have been continuously evolving towards higher performance, larger memory sizes, larger memory bandwidths and relatively lower costs. This high computing power mainly arises from a fully pipelined and highly parallel architecture, with extremely high memory bandwidths.

The NVIDIA® Tesla™ C1060 computing processor enables the transition to energy efficient parallel computing power by bringing the performance of a small cluster to a workstation. With 240 processor cores and a standard C compiler that simplifies application development, Tesla scales to solve the world’s most important computing challenges more quickly and accurately.
The GeForce 9400 GTX architecture has 16 stream processors and access to 512Mb of RAM. The theoretical performance of this card is 44 Gflops. Although currently the GPUs in this setup are low end, the approach detailed here will also be applicable to high-end and future devices.

3.5 MEMETIC ALGORITHM ON GPU

Parallel Memetic algorithm has been constructed using the MATLAB m-code programming language. The GPU is especially well-suited to address problems that can be expressed as data-parallel computations; one of the most important things we can do to prepare for GPU computing with MATLAB is to understand those segments of our target application where data parallel computations take place. This is our first indication of place in our code that is prospects for GPU computing.

Next, profiling our application to identify the segments of our code that represent the most time consuming regions will provide further indication of those segments of our code that could benefit from GPU computing. The MATLAB Profiler tool helps tremendously in determining where best to focus your energy when moving code to the GPU. Looking at the results of the profiler, a user can determine where the program is spending most of its time and focus transformation time to the area of code to get the biggest return.

GPUtil, developed by the GP-You group, allows Matlab code to get benefited from the computational power of modern GPUs. It is built on top of NVIDIA CUDA. The acceleration is transparent to the user, only the declaration of variables needs to be changed using new GPU-specific keywords. Algorithms need not be changed. A wide range of standard Matlab functions have been implemented. GPUtil is available as freeware for Windows and Linux from the GP-You download page.
GPUmat uses a technology developed by NVIDIA called CUDA SDK which allows programming the GPU for general purpose applications. The GPUmat core is based on CUDA libraries, such as CUFFT and CUBLAS, and many other functions developed and optimized by the GP-you Group for the GPU architecture.

3.6 EXPERIMENTAL RESULTS

In this section, the quality of the bisections produced by GPU based parallel Memetic algorithm are analysed with experiments on a large number of hypergraphs that are part of the widely used ISCAS circuit partitioning benchmarks suite. All experiments were carried out on Pentium core 2 Quad processor 2.6 GHz with NVIDIA Tesla C1060 computing processor and GeForce GTX 9400 GPU display card, with 8GB main memory and 512MB GPU memory.

The system might have more than one GPU installed. By default GPUstart selects the first available GPU device. GPU based implementation was compared with software implementation running on single CPU. The following parameters are used in the experiments: For the Memetic Algorithm, the population size was set to 10, the probability for crossover was 0.95, and the probability for mutation was 0.05 for all test problems as it was the best configuration found empirically for the Genetic Algorithm. Table 3.1 shows the statistics for the experiment. The results obtained from hmetis is listed in column 4 and 5. The speedup obtained using a single GPU card is listed in Column 12. The speedup obtained is on an average of 3.89×. By using the NVIDIA Tesla C1060, the available global memory increases by 8GB. Column 11 lists the runtimes for Tesla C1060. The speedup obtained against the commercial tool in this case is listed in Column 13. The speedup obtained in this case is on an average of 5.202×. Figure 3.4, 3.5 and 3.6 shows the comparison of speedup, timing and cutsize value of benchmark circuits.
obtained through running MA on CPU, GPU GTX9400 and TeslaC1060 respectively.

**Table 3.1 Performance of MA for VLSI Partitioning Problem**

<table>
<thead>
<tr>
<th>ISCAS Benchmark circuit</th>
<th>Number of cells</th>
<th>Number of nets</th>
<th>hmetis</th>
<th>MA on CPU Cut</th>
<th>T(S)</th>
<th>MA on GTX 9400 GPU card Cut</th>
<th>T(S)</th>
<th>MA on Tesla C1060 GPU processor Cut</th>
<th>T(S)</th>
<th>Speed up GTX 9400</th>
<th>Tesla C1060</th>
</tr>
</thead>
<tbody>
<tr>
<td>S298</td>
<td>136</td>
<td>130</td>
<td></td>
<td>12</td>
<td>30</td>
<td>19</td>
<td>123</td>
<td>11</td>
<td>30</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>S386</td>
<td>172</td>
<td>165</td>
<td></td>
<td>27</td>
<td>50</td>
<td>36</td>
<td>163</td>
<td>28</td>
<td>50</td>
<td>26</td>
<td>40</td>
</tr>
<tr>
<td>S641</td>
<td>433</td>
<td>410</td>
<td></td>
<td>14</td>
<td>300</td>
<td>45</td>
<td>1868</td>
<td>15</td>
<td>300</td>
<td>13</td>
<td>200</td>
</tr>
<tr>
<td>S832</td>
<td>310</td>
<td>291</td>
<td></td>
<td>35</td>
<td>500</td>
<td>55</td>
<td>1055</td>
<td>39</td>
<td>500</td>
<td>37</td>
<td>400</td>
</tr>
<tr>
<td>S953</td>
<td>440</td>
<td>417</td>
<td></td>
<td>46</td>
<td>200</td>
<td>96</td>
<td>618</td>
<td>45</td>
<td>200</td>
<td>43</td>
<td>150</td>
</tr>
<tr>
<td>S1196</td>
<td>561</td>
<td>547</td>
<td></td>
<td>79</td>
<td>250</td>
<td>123</td>
<td>375</td>
<td>75</td>
<td>250</td>
<td>74</td>
<td>200</td>
</tr>
<tr>
<td>S1238</td>
<td>540</td>
<td>526</td>
<td></td>
<td>85</td>
<td>150</td>
<td>127</td>
<td>397</td>
<td>79</td>
<td>150</td>
<td>77</td>
<td>104</td>
</tr>
<tr>
<td>S1488</td>
<td>667</td>
<td>648</td>
<td></td>
<td>86</td>
<td>454</td>
<td>104</td>
<td>1238</td>
<td>80</td>
<td>454</td>
<td>78</td>
<td>406</td>
</tr>
<tr>
<td>S1494</td>
<td>661</td>
<td>642</td>
<td></td>
<td>80</td>
<td>500</td>
<td>120</td>
<td>1345</td>
<td>75</td>
<td>500</td>
<td>74</td>
<td>410</td>
</tr>
<tr>
<td>S2081</td>
<td>122</td>
<td>121</td>
<td></td>
<td>14</td>
<td>150</td>
<td>50</td>
<td>354</td>
<td>13</td>
<td>150</td>
<td>12</td>
<td>104</td>
</tr>
<tr>
<td>S5378</td>
<td>2994</td>
<td>2944</td>
<td></td>
<td>145</td>
<td>854</td>
<td>171</td>
<td>5201</td>
<td>151</td>
<td>854</td>
<td>140</td>
<td>590</td>
</tr>
<tr>
<td>S9234</td>
<td>5845</td>
<td>5822</td>
<td></td>
<td>185</td>
<td>1723</td>
<td>231</td>
<td>9654</td>
<td>191</td>
<td>1723</td>
<td>180</td>
<td>1500</td>
</tr>
<tr>
<td>S13207</td>
<td>8652</td>
<td>8530</td>
<td></td>
<td>305</td>
<td>1874</td>
<td>340</td>
<td>9789</td>
<td>311</td>
<td>1874</td>
<td>300</td>
<td>1604</td>
</tr>
<tr>
<td>S15850</td>
<td>10384</td>
<td>10296</td>
<td></td>
<td>395</td>
<td>985</td>
<td>421</td>
<td>8534</td>
<td>411</td>
<td>985</td>
<td>390</td>
<td>685</td>
</tr>
<tr>
<td>Avg.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3.897</td>
<td>5.202</td>
</tr>
</tbody>
</table>
Figure 3.4  Speedup Comparison

Figure 3.5  Timing Comparison
3.7 CONCLUSION

In this chapter, a parallel MA has been implemented on consumer-level graphics cards in VLSI Circuit bi-partitioning problem to optimize cutsize value with balance constraints. Algorithm was tested on large number of hypergraphs that are part of the widely used ISCAS circuit partitioning benchmarks suite. The problem has been simulated separately on CPU, GPU cards. Using the Genetic control parameters, simulation experiments were conducted to investigate the capability of the algorithm’s problem solving qualities in terms of quality of solution and execution time on CPU and GPU cards. It was found that the Memetic Algorithms took more execution time when running on CPU than the parallel MA running on GPU to generate similar results. On comparison of the results obtained by running on CPU and GPU implementations, it was found that GPU implementation produced better results.