A SCALABLE EVOLUTIONARY ALGORITHM FOR SOLVING THE ONE-DIMENSIONAL BIN PACKING PROBLEM ON GPU USING CUDA

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

SÜKRÜ ÖZER ÖZCAN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

SEPTEMBER 2015

Approval of the thesis:

A SCALABLE EVOLUTIONARY ALGORITHM FOR SOLVING THE ONE-DIMENSIONAL BIN PACKING PROBLEM ON GPU USING CUDA

submitted by **SÜKRÜ ÖZER ÖZCAN** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Gülbin Dural Ünver _____
Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. Adnan Yazıcı _____
Head of Department, **Computer Engineering**

Prof. Dr. Ahmet Coşar _____
Supervisor, **Computer Engineering Department, METU**

Asst.Prof.Dr.Yusuf Sahillioğlu _____
Co-supervisor, **Computer Engineering Department, METU**

Examining Committee Members:

Prof. Dr. İsmail Hakkı Toroslu
Computer Engineering Dept., METU _____

Prof. Dr. Ahmet Coşar
Computer Engineering Dept., METU _____

Asst.Prof.Dr. Yusuf Sahillioğlu
Computer Engineering Dept., METU _____

Prof. Dr. Faruk Polat
Computer Engineering Dept., METU _____

Assoc. Prof. Tansel Özyer
Computer Engineering Dept., TOBB ETU _____

Date: _____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: SÜKRÜ ÖZER ÖZCAN

Signature :

ABSTRACT

A SCALABLE EVOLUTIONARY ALGORITHM FOR SOLVING
THE ONE-DIMENSIONAL BIN PACKING PROBLEM
ON GPU USING CUDA

Özcan, Sükrü Özer
M.S., Department of Computer Engineering
Supervisor : Prof. Dr. Ahmet Coşar
Co-Supervisor : Asst.Prof.Dr.Yusuf Sahillioğlu

September 2015, 105 pages

One-dimensional Bin Packing Problem (1DBPP) is a challenging NP-Hard combinatorial industrial engineering problem which is used to pack finite number of items into minimum number of fixed size bins. Different versions of 1DBPP can be faced in real life. Although the problems that have small number of items up to 20 can be solved with brute-force algorithms, large problem instances of the 1DBPP cannot be solved exactly due to its intractable nature. Therefore, heuristic approaches such as Genetic, Particle Swarm, Tabu Search, and Minimum Bin Slack have been widely used to solve this important problem (near-) optimally. Most of the the state-of-the-art algorithms that have been proposed to solve the 1DBPP are executed on a single processor and do not make use of the high performance opportunities that are offered by the recent parallel computation technologies. In this study, we increase the performance of a Grouping Genetic Algorithm (GGA) by harnessing the power of the graphics processing unit (GPU) using Compute Unified Device Architecture (CUDA) for the first time in literature. The time consuming crossover and mutation processes of the GGA are executed on the GPU and the population of solutions is kept on the global memory of GPU while running the whole algorithm in a heterogeneous computing environment. The obtained experimental results on 1,238 benchmark problem instances show that the proposed algorithm, CUDA GGA for 1DBPP (CUDA-GGA-1DBPP), is a high performance and scalable algorithm that can be counted among the

best performing algorithms in literature and it is about 60 times faster than its CPU counterpart.

Keywords: 1D Bin packing, grouping genetic, CUDA, GPU

ÖZ

TEK BOYUTLU KUTU PAKETLEME PROBLEMİNİN GRAFİK İSLEMCİ ÜZERİNDE CUDA KULLANILARAK ÖLÇEKLENEBİLİR EVRİMSEL ALGORİTMA İLE ÇÖZÜMÜ

Özcan, Sükrü Özer
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi : Prof. Dr. Ahmet Coşar
Ortak Tez Yöneticisi : Yrd.Doç.Dr. Yusuf Sahillioğlu

Eylül 2015 , 105 sayfa

Farklı genişlik ve uzunluktaki nesnelerin mümkün olan en az sayıda sabit kapasiteli kutu kullanılarak yerleştirilmesini amaçlayan tek boyutlu kutu paketleme problemi, çözümü polinom zamanlı olmayan (NP) ve endüstri mühendisliği alanında popüler olan kombinatoriyal bir en iyileme problemdir. Çok bilinen bu endüstri probleminin çok amaçlı versiyonlarıyla gündelik hayatta sıkça karşılaşılmaktadır. 20 kutuya kadar olan küçük olçekli problemler karmaşık olmayan basit algoritmalar kullanılarak çözülebilse de, büyük ölçekli problemlere optimal çözüm üretmek problemin kolay takip edilemeyen doğası sebebiyle mümkün olmayabilir. Kesin çözüm bulmanın mümkün olmadığı büyük ölçekli söz konusu problemlere en iyi çzümü bulabilmek amacı ile şimdiye kadar En Az Boşluk Algoritması, Tabu Araması, Parçacık Sürü Optimizasyonu benzeri sezgisel yontemler yaygın olarak kullanılmıştır. Ancak, problemin çözümüne yönelik geliştirilen bu sezgisel yöntemler performansı artıran son nesil paralel bilgisayar teknolojisi kullanılmaksızın tek bir işlemci kullanilarak çözülmüştür. Bu çalişma ile literatürde ilk defa CUDA (Compute Unified Device Architecture) kullanılarak grafik islemci birimi (GPU) katkısı ile Gruplama Genetik Algoritması (GGA)'nın performansı artırılmıştır. Heterojen bir mimari üzerinde koşturulan program ile GGA'nin islem süresini uzatan çarprazlama ve mutasyon algoritmalari GPU uzerinde çalıştırılmış, sonuçlar ise ekran kartı üzerinde bulunan global bellek üzerinde tutulmuştur. 1.238 farkli problem seti uzerinde koşturulan problem ve sonuçlari, Tek

Boyutlu Kutulama CUDA GPU algoritmasının literatürde yer alan en iyi algoritmalar arasında yer alabileceğini ve CPU karşılığına göre de 60 kat daha hızlı olabileceğini göstermiştir.

Anahtar Kelimeler: 1B Kutu paketleme, gruplayan genetik, CUDA, GPU

To my family and especially my little princess who is in heaven

ACKNOWLEDGMENTS

Many thanks to my supervisor, Prof. Dr. Ahmet Coşar, for spending his months and effort during this study. It has been a great privilege to have been mentored by such an esteemed researcher.

Special thanks to Dr. Tansel Dökeroğlu who provided encouragement for this study and also for his valuable insight and his guidance about this thesis.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

1DBPP	One-Dimensional Bin Packing Problem
ACO	Ant Colony Optimization
ALU	Arithmetic Logic Unit
BF	Best Fit
BFB	Best Fit Bin
BFD	Best Fit Decreasing
CUDA	Compute Unified Device Architecture
CUDA-GGA-1DBPP	CUDA Grouping Algorithm for 1DBPP
CPU	Central Processing Unit
DRAM	Dynamic Random Access Memory
EA	Evolutionary Algorithm
FFD	First Fit Decreasing
FF	First Fit
NF	Next Fit
GA	Genetic Algorithm
GGA	Grouping Genetic Algorithm
GGA-CGT	Grouping Genetic Algorithm with Controlled Gene Transmission
GFA	Gravitation Field Algorithm
GPC	Graphics Processing Unit
HPC	High-Performance-Computing
H-SGGA	Hybrid Steady State GGA
MBS	Minimum Bin Slack
MISD	Multiple Instruction Single Data
MIMD	Multiple Instruction Multiple Data
MTRP	Martello and Toths Procedure
SAW	Sufficient Average Weight
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Thread
SM	Streaming Multiprocessor
SISD	Single Instruction Single Data
TDR	Timeout Detection Recovery
TPC	Texture Processing Unit
TSP	Traveling Salesman Problem
VRP	Vehicle Routing Problem

CHAPTER 1

INTRODUCTION

One-dimensional Bin Packing Problem (1DBPP) is a challenging NP-Hard combinatorial industrial engineering problem which is used to pack finite number of items into minimum number of bins. Different versions of 1DBPP may be faced frequently in real life situations [2, 3, 5, 7, 8]. Although the problems that have small number of items up to 20 can be solved with brute-force algorithms, large problem instances of the 1DBPP cannot be solved exactly due to its intractable nature. Therefore, heuristic approaches such as Genetic, Particle Swarm, Tabu Search, and Minimum Bin Slack have been widely used to solve this important problem (near-) optimally. Most of the the state-of-the-art algorithms that have been proposed to solve the 1DBPP are running on a single processor and do not make use of the high performance opportunities that are offered by the recent parallel computation technologies. In this study, we increase the performance of a Grouping Genetic Algorithm (GGA) by making use of the power of the graphics processing unit (GPU) using Compute Unified Device Architecture (CUDA) for the first time in literature. The time consuming crossover and mutation processes of the GGA are executed on the GPU environment and the population of solutions is kept on the global memory of GPU while running the whole algorithm in a heterogeneous computing environment.

1.1 1D Bin Packing Problem

The bin packing problem (BPP) is an NP-hard combinatorial optimization problem which is used to pack finite number of items of different volume [9]. The general

purpose of the problem is to pack items of interest subject to various constraints, such that the overall number of bins is minimized .The BPP is the process of packing N items into bins which are unlimited in numbers and same in size and shape. The bins are assumed to have a capacity of $C > 0$, and items are assumed to have a size S_i for I in $\{1, 2, ..., N\}$ where $S_i > 0$. The goal is to find minimum number of bins in order to pack all of N items. Formal mathematical formulation of the problem can be written as following:

$$\text{minimize } z = \sum_{i=1}^{n} y_i$$

$$\text{subject to} \quad \sum_{i=1}^{n} w_j x_{ij} \leq c y_i, \qquad i \in N = \{1, ..., n\},$$

$$\sum_{i=1}^{n} x_{ij} = 1, \qquad j \in N,$$

$$y_i = 0 \text{ or } 1, \qquad i \in N,$$

$$x_{ij} = 0 \text{ or } 1, \qquad i \in N, j \in N,$$

where w_i is the weight of item i,

$$y_i = \begin{cases} 1 & \text{if bin } i \text{ is used;} \\ 0 & \text{otherwise,} \end{cases}$$

$$x_{ij} = \begin{cases} 1 & \text{if item } j \text{ is assigned to bin } i; \\ 0 & \text{otherwise,} \end{cases}$$

There are many variations of the problem such as one dimensional, 2D packing, linear packing, packing by cost, packing by weight, and so on. One dimensional bin packing problem is chosen as the main focus of our study. In one dimensional BPP objects have a single dimension such that cost, weight, time, size, etc... In this variation finding the optimal solution is also known to be NP-hard. The problem can be modeled as an integer programming problem for relatively small sizes without degrading the computational time. For larger problems, different heuristic methods have been developed in order to find optimal solution and to deal with run-time performance. In addition to strength of formulation and power of algorithm, hardware capabilities can also be utilized in order to improve run time performance. In this

study, we utilize CUDA platform in order to analyze contribution of GPU on run time performance of BPP solving with Genetic Algorithm. Genetic algorithms and its variations, First-Fit (FF) algorithm, Next-Fit (NF) algorithms, Best-Fit (BF) Algorithm, Next-Fit Decreasing (NFD), Best-Fit Decreasing (BFD) are some of the well- known algorithms in the literature to solve BPP. Time complexity of the algorithms can be seen in Table 1.1:

Table 1.1: Time Complexity of BPP Algorithms

Algorithms	NF	FF	BF	NFD	FFD	BFD
Time Complexity	O(n)	O(nlogn)	O(nlogn)	O(nlogn)	O(nlogn)	O(nlogn)

1.2 Genetic Algorithms as Evolutionary Optimization Tools

Genetic Algorithms (GAs) are evolutionary heuristic search algorithms of machine learning which are inspired by natural evolution and genetics. Reproduction, mutation, recombination and selection are key mechanisms of Evolutionary Algorithms (EA) that are used to solve optimization problems since they represent an intelligent exploitation of a random search. Although known as nonlinear Monte Carlo methods to sample from complex high-dimensional search space, GAs harness the resources and simulate the natural selection; "survival of the fittest", and stands as a promising methods for exponentially growing NP-hard optimization problems [4, 10, 11, 13, 14]. Bin Packing, Traveling Sales Person and Quadratic Assignment Problem [15, 16, 17] are well-known problems that are solved with EAs.

Evolution starts with the creation of a set of character strings that are analogous to the base-4 chromosomes which represents the population of individuals as we see in our own DNA. The individuals in the population then go through a process. Different than natural evolution, ongoing process is managed purposefully and directed to choose the most promising offspring in machine learning. The encoding takes place during the reproduction and the crossover operation enables the selection. Some GAs use a simple method called as fitness-proportionate selection of the fitness measure to select individuals probabilistically. Other implementations use a model called as tournament selection in which certain randomly selected individuals in a subgroup compete and the fittest is selected. By the way, mutation plays a key role in this pro-

cess and GAs use a stochastic process, which provides better results than random, for the mutation.

GAs are developed in a cyclic manner and each cycle is called as a generation. The process starts over a randomly generated population and then old population is discarded from the process with the generation of new population.

The Grouping Genetic Algorithms (GGA), an extension of the conventional Genetic Algorithms, were developed by Falkenauer to solve clustering problems [18, 19, 20]. In GGA, the chromosomes are enhanced with a group element containing the group composition and all operators of the GAs work on group elements of chromosomes. As a well-known NP-hard grouping problem, 1DBPP, is one of the application area of GGAs [21].

The GGAs gain a power over the standard GAs by reducing the selection pressure. Since the chromosomes are grouped and the operators are applied to groups rather than a single gene, algorithms converges the solution faster then the classical approach. Grouping approach provides a room for simultaneous run which reduces the number of iterations and so time required to reach optimal solution. Different selection mechanisms and the genetic operators are very crucial to the success of GA [22, 23]. So, a benchmark of selection schemes are still needed to study in terms of finding the most effective approach in terms of convergence velocity. However, GGAs are applicable only for complex problems that can be divided into sub-problems without any overlap and then can be combined back to original problem to generate the solution (see Fig. 1.1 for the flowchart of GGAs). With the help of additional effort and caution, GGAs stand as a promising approach to produce efficient solutions to NP-Hard problems.

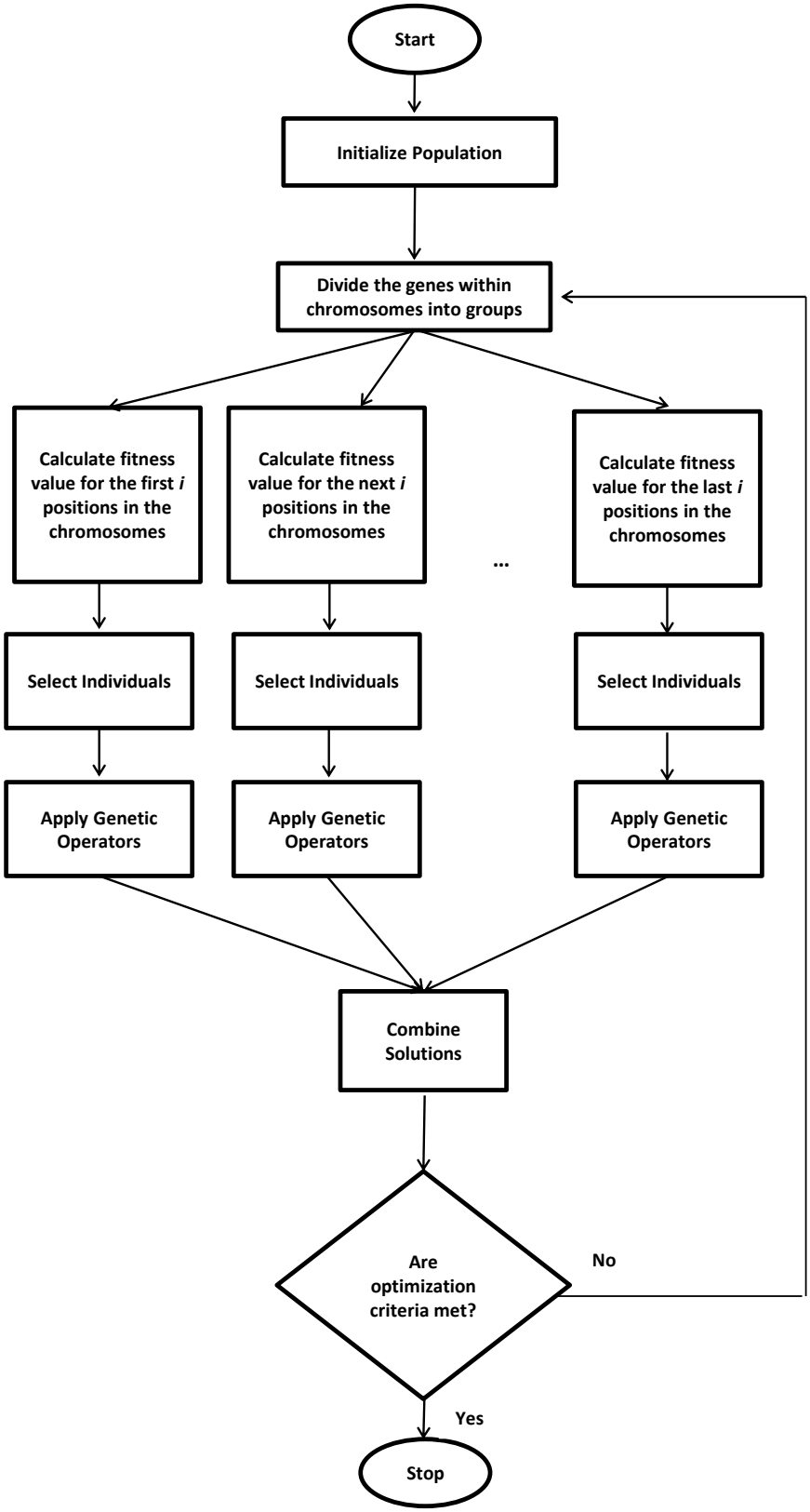

Figure 1.1: Grouping Genetic Algorithm Flowchart [21]

1.3 Compute Unified Device Architecture

Especially for the last three decades, increase in the clock speed of the processors give a way to the search of looking alternative ways to have a better computational power. As a result, CPU manufacturers began to sell processors with more than one cores rather than a single core. These tendency named as multi-core revolution.

With multi-core revolution the modern parallel computation environments have found a thorough interest in the field due to their efficiency and accuracy on hard problems. Within a single chip multi-core processors host many cores and provide higher computing power. These cores can be used by parallel algorithms and they can increase the quality of the results and reduce the time required to get the results of NP-Hard problems such as 1DBPP [53].

While CPUs are evolving, graphically driven operation systems have emerged from urgent need for additional display accelerators in personal computers. In 1980s a company named Silicon Graphics study the idea to have 3D graphics and released OpenGL library. After that, games, which have 3D Graphics, began to work on to offer realistic 3D environment. The release of GPUs capable of implementing Microsoft's DirectX 8.0 in 2001 was a one of the most important breakthrough in development of programmable graphic units. In the early 2000s, GPUs produce colors for every pixel via pixel shaders which are programmable arithmetic units. Than researches focused on putting any data to the pixel shaders other than just colors. And they faked the GPUs with sending back numerical data other than colors after processed in the pixel shaders. Initial results from the studies brought hope to use GPUs which have high arithmetic throughput for computational studies.

While gaming industry pushes the limitations of the GPUs, limited number of developers were trying to learn OpenGL or DiretX funcitons which were only way to communicate with GPU. In addition to that they were dealing to locate their data on the arbitrary GPU memory even in a simple computation. After approximately five years later, NVIDIA announce the GeForce 8800GTX with DirectX 10. This GPU was the first one built with Compute Unified Device Architecture (CUDA) Architecture.

CUDA copped with the early computable GPUs drawbacks and designed with a unified shader pipeline permits every arithmetic logic unit (ALU) on the chip to be used by programs for arithmetic computations.No later than a few months after the announcement, NVIDIA made a public compiler for the CUDA language which has limited number of keywords and standard C language [24].

Development in both High-Performance-Computing (HPC) and CUDA which is a parallel computing platform and programming model, have led to a fundamental paradigm shift in parallel programming.

According to Flynn's Taxonomy, computer architecture comprises of four different types corresponds to how instructions and data flow through cores including:

- Single Instruction Single Data (SISD)
- Single Instruction Multiple Data (SIMD)
- Multiple Instruction Single Data (MISD)
- Multiple Instruction Multiple Data (MIMD)

SISD refers to the traditional computers which has only one core, and simply means serial architecture. SIMD is a parallel architecture while all cores execute the same instructions but different data. MISD is a special purpose architecture where all cores uses the same data with different instruction. In MIMD architecture different cores operates with different data.

In CUDA Architecture, it can be referred as Single Instruction, Multiple Thread (SIMT), where the same instruction can be operated in multiple threads, because of the design differences between GPU and CPU. A CPU core, relatively heavy-weight, is designed for very complex control logic, seeking to optimize the execution of sequential programs, while a GPU core, relatively light-weight, is optimized for data-parallel tasks with simpler control logic, focusing on the throughput of parallel programs. These difference causes us to change from traditional programming to light-weight programming which is highly and repeatedly optimized. Since every procedure of our algorithm can't run on GPU due to its limited control logic, we used heterogeneous computing techniques.

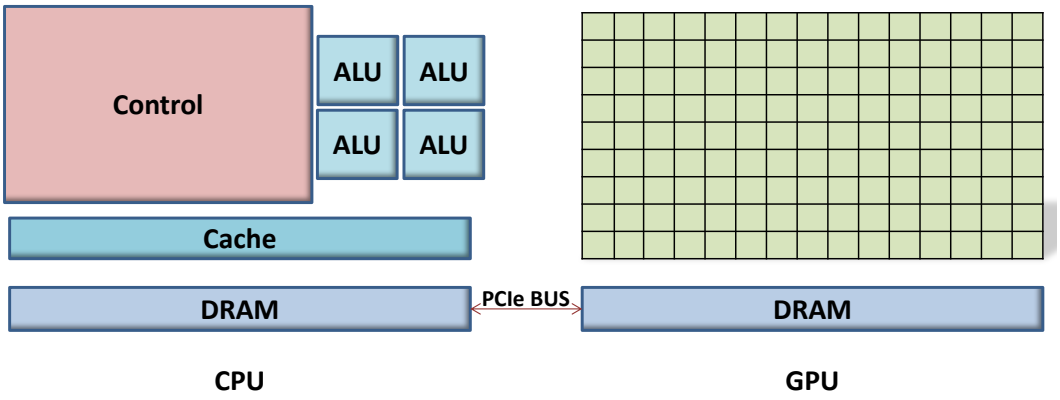

Figure 1.2: Heterogeneous Architecture [6]

Heterogeneous computing is a type of computing which applying tasks to the GPU or CPU choosing the one best suits for them. Main actor in this architecture is PCI-Express bus which connects CPU and GPU as shown in Figure 1.2 . CPU is referred as *host*, while GPU is a *device* in this architecture. We will use these terms from now on in our study.

In addition to the Figure 1.2, the simple architecture of the device can be listed as following;

- A host interface that connects the GPU to the PCI Express bus
- Copy Engines
- A DRAM interface that connects the GPU to its device memory
- Some number of TPCs or GPCs (texture processing clusters or graphics processing clusters), each of which contains cache and some number of streaming multiprocessors (SMs)

The program execution starts on host, some of the procedures are executed in GPU an then solutions are displayed in host. Device capability can be defined with *Number of CUDA cores* and *Memory Size*, and device performance can be defined with *Peak Computational Performance* (tflops (tflops-trillion floating-point calculations per second) and *Memory Bandwidth* (gigabytes per second, GB/s.) . Our CUDA device (GeForce 750 Ti) characteristics comparing some other NVIDIA's CUDA devices capabilities and performances can be shown in Figure 1.3 [24].

	GeForce GTX 750 Ti	Quadro K5200	Tesla K80
CUDA Cores	640	2304	2496 x2
Memory	2GB GDDR5	8GB GDDR5	12GB x2 GDDR5
Peak Performance	1.306 Tflops	2.1 Tflops	8.74 Tflops
Memory Bandwidth	86.40 GB/s	192GB/s	240 GB/s x2

Figure 1.3: Device Capabilities and Performances

A CUDA enabled device is well-know with solving highly parallel and compute-intensive computation because it is designed with more transistors to process data rather than data caching and flow control. A detailed version of our CUDA Device(Maxxwell GM107) Architecture is shown in Figure 1.4

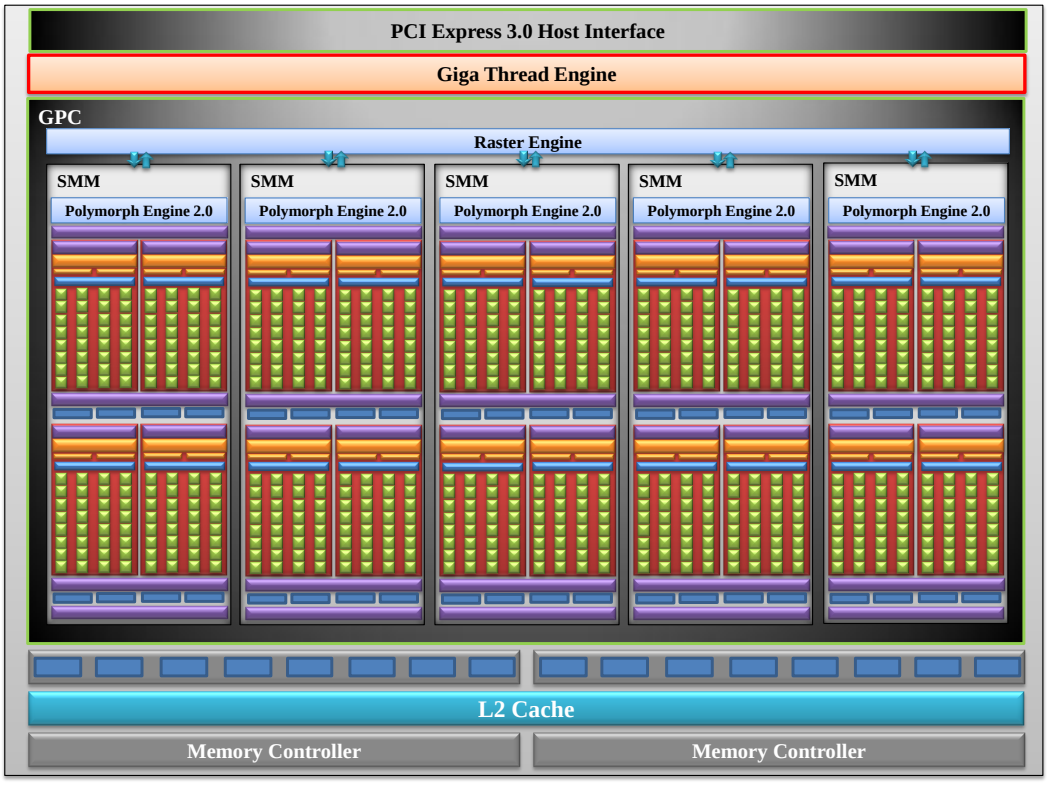

Figure 1.4: Maxxwell GM107 Device Architecture [26]

Our device has five Streaming Multiprocessors, referred as SMM. SMM uses a quadrant-

based design with four 32-core processing blocks each with a dedicated warp scheduler capable of dispatching two instructions per clock. Each SMM provides eight texture units, one polymorph engine (geometry processing for graphics), and dedicated register file and shared memory.

CUDA is a software development platform consisting of CUDA-accelerated libraries, OpenACC like compiler directives, and extensions of some of the standard programming languages such as C, C++ and Fortran. C/C++ use "nvcc" compiled code, or NVIDIA's LLVM-based C/C++ compiler. Fortran can also be used via CUDA Fortran based on PGI CUDA Fortran developed by The Portland Group.

In addition to above, OpenCL, MS DirectCompute, OpenGL Compute Shaders and C++ AMP interfaces are also supported. Wrappers for Python, Java, Ruby, Lua, Haskell, R, MATLAB, and others also exist.

GPUs are used in computer games for graphics rendering as well as in calculating game physics (e.g. smoke, fire, fluids effects). CUDA, has also been in computational biology, cryptography, and other fields in addition to computer graphics, providing speed-ups of 10 times or more.

In this study, our main purpose is to solve offline 1DBPP using GGAs with the help of massive parallel execution capable GPU and CUDA Software Platform.

One of the processes need to be done in CUDA software development platform is allocating memory on device side and copying the data from host to device in order to use it in kernels. CUDA has fallowing command to allocate memory on GPU;

```
cudaMalloc((void ) g population, 10 size of population
    sizeof(chromosome));
```

If any data is need to be used in device code, it must be copied with *cudaMemcpy* . For example we need the item values, which we read from data set, on device. So, we copy them with the following command;

```
cudaMemcpy(g item values, hitemValues, 500 sizeof(int),
    cudaMemcpyHostToDevice);
```

First parameter "*g_item_values*" is the device pointer where the items copied, "*hitem-*

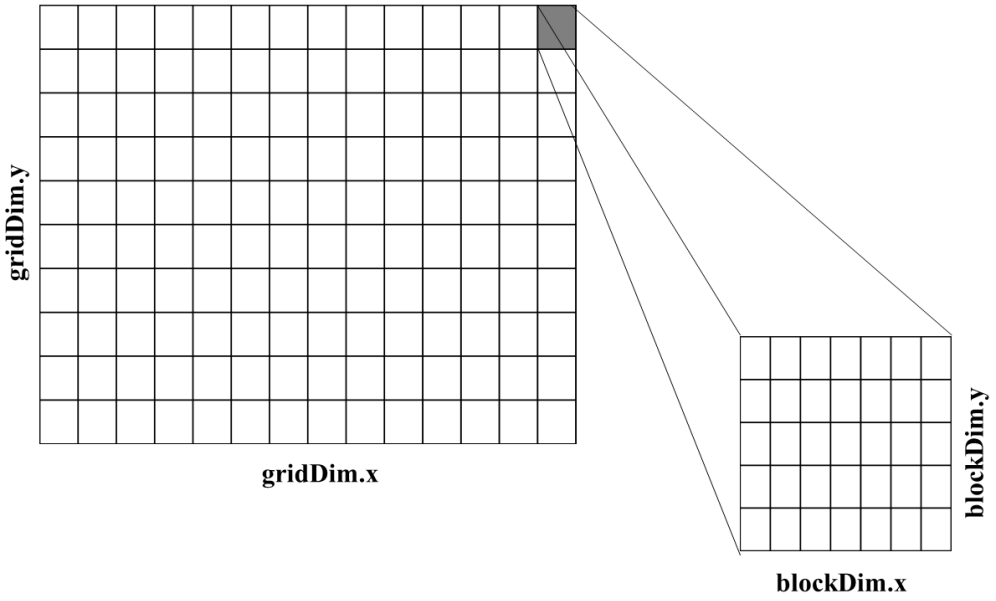

Figure 1.5: 2D Grid and Thread Block

Values" is the source values, third parameter is the size of the data to be copied and the final parameter defines the direction which the copy operation occurs. *cudaMemcpyHostToDevice* defines the copy operation take data from host and paste it to the device pointer where *cudaMemcpyDeviceToHost* makes the opposite direction.

Kernels are pieces of functions which run on GPU in GPU related terms. For launching a kernel some kernel execution parameters which must be inside triple-angle-brackets and an argument list are needed;

setup_kernel < < < 64, 64 > > > (devState or devMRGStates);

generate_normal_kernel < < < 64, 64 > > > (devState or devMRGStates , 1000, dev_rnd_Results);

With the execution parameters, the configuration of how the kernels are going to work is determined. Kernels are launched as *grids of blocks of threads* . The first value *64* is simply the number of blocks to launch, grid dimension at the same time, the second value *64* is the number of threads within each block. These terms are important because the performance change is observed when we tune them. General relationship between grid, block and thread is shown as 2D and 3D in Figures 1.5 and 1.6 respectively.

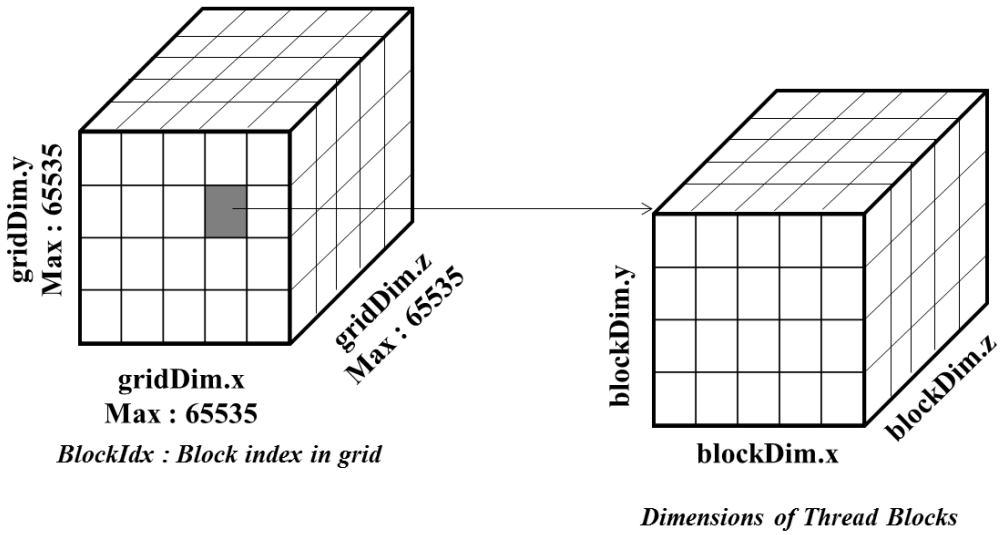

Figure 1.6: 3D Grid and Thread Block

Threads are a special case in CUDA platform. They can communicate with each other over a shared memory which is defined usually at the beginning of a kernel with the " __*shared* " pre-word. For example:

Every kernel needs a unique id to be distinguished by the other processes. These unique ids can be generated differently depending on the launching configuration.

The following code is general lines to generate a unique id in each concurrent process:

```
const unsigned long long int tid = blockIdx.x                    //1D
+ blockIdx.y * gridDim.x                                         //2D
+ gridDim.x * gridDim.y * blockIdx.z;                            //3D
```

For example, in the configuration of the kernel of generating initial population; there is only one dimension and the unique id is simply *blockIdx.x* because other all variables are zeros.

Each concurrent process has a unique identification number to be distinct. This id number can be generated with sum of threadIdx.x with product of blockIdx.x and blockDim.x if DIM3 configuration is not used. In addition to that shared memory size which is the size of all variables used by threads in the same thread block and streaming attributes can be defined in this configuration parameters. Detailed infor-

mation can be found in NVIDIA's CUDA Toolkit Documentation [25] .

In this study we implemented CUDA Streams, for limited speed up and practice. CUDA streams provides four types of concurrency to get even faster;

- CPU/GPU concurrency
- Host-Device Memory Copy Concurrency
- Kernel Concurrency
- Multi-GPU Concurrency

Since CPU and GPU operations can be independent from each other, they can be run asynchronously. Another exhausting process is the host to device memory copy and it can be run asynchronously too. Moreover, current GPUs have a computing capability which allows to be run 4 kernels concurrently. Final one as it can be obviously seen from it's name, Multi-GPUs can be run concurrently.

CHAPTER 2

RELATED WORK

In this chapter, we give information about the state-of-the-art methods for solving the 1DBPP, GGAs, and recent studies about the optimization of other combinatorial optimization problems that are solved by using CUDA.

2.1 State-of-the-art 1DBPP Algorithms

Many exact and approximation algorithms can be found in the literature about BPP. Martello and Toth's book named "Knapsack Problems" in which has a section about BPP, explained briefly about approximation algorithms like NF algorithm which simply assigns the next item to the current bin after putting first item to the first bin [41]. If it doesn't fit to the current bin, it opens a new bin and put the item in it. A better approximation algorithm is the FF where putting the next item to the bin it fits beginning from the first bin. If it can't find any bin to fit, it opens a new bin and assigned the item to the new bin. BF algorithm is a version of FF and it put the next item to the feasible bin which has smallest residual capacity. Time complexity for NF is *O(n)* where FF and BF's are *O(nlogn)* if it stores the slacks of the bin in the leaves. If the algorithm sorts the items in decreasing order, the algorithm's names change as Next-Fit Decreasing (NFD), First-Fit Decreasing (FFD) and Best-Fit Decreasing (BFD) respectively and their time complexities become the same as *O(nlogn)*. In addition the the approximation algorithms, they provided an exact algorithm named (MTP) and a resulting reduction procedure named (MTRP).

Scholl et al. developed an hybrid solution, named BISON, consist of several heuristics

[48]. They found tabu search as the most effective procedure among other heuristics. In addition to that they provided a new local lower bound method which is well-known in capacity related problems.

Schwerin and Wascher criticized the results provided by the early studier of BPP because of the test design they use [49]. Schwerin and Wascher found the results unsatisfactory and unconvincing. Because of that, they provided a new problem generator for the BPP.

A new model and an exact procedure is also presented in 1999 by Valério de Carvalho. The procedure combines column generation and branch-and-bound and it grows in two dimension [34].

With respect to FFD and BFD, a new and effective algorithm called Minimum Bin Slack (MBS) has been produced by Gupta and Ho in 1999 [39]. Their algorithm generates optimal solutions if all items' total weight is less than or equal to twice the bin size. Random generator is used to build the test data sets. They named a difficult set of problems with variable bin capacities and 5 new data sets. For that 5 difficult problems sets, their algorithm produced one total bin less than FFD and BFD each.

Based on MBS, a new procedure named MBS$'$ is designed by Fleszar and Hindi in 2002 [7]. The procedure's specialty is putting one item to the next bin to fill before the classical MBS's search function run. In addition to MBS$'$, three new procedure is designed using MBS$'$. First one is relaxed MBS$'$, which permits non-zero residual capacity in bins, second one is perturbation MBS$'$ which aims to have a better solution just using one-packing function and building new bins, third one is sampling MBS$'$ which calls MBS$'$ recursively only changing the order of items randomly. The only drawback in this study is computational complexity of the procedures O(2^n) .

Alvim et al. (2004) put forward an hybrid heuristic which has several characteristics including tabu search, differencing and unbalancing, getting initial solution from dual-min-max problem, lower bound method and so on [1]. Their most effective contribution came from tabu search improvement procedure. The benchmark data set results were better in 41 instances than Fleszar and Hindi [7]. Fleszar and Charalambous see the problem of general classical bin-oriented heuristics which is putting

small items mostly to the initial bins means bigger slack at later bins [8]. To deal with that problem, a control mechanism named Sufficient Average Weight (SAW) also the prefixes of the procedures is designed. In the beginning of the procedure, they took the average weight of the remaining items. When constructing a new bin, the solution compares the current bin average weight with the remaining items' average weights. If it passes the test of being sufficient, current bin construction will be approved. They integrated this SAW control with FFD,B2F,MBS and MBS′ changing their names as SAWB2F, SAWMBS, SAWMBS′. In addition to the procedures above they also used SAW with Perturbation MBS′ resulting in solution quality perspective, the best of the known improvement heuristics. Plus no increase in computational time [8]

A different simulated annealing heuristic is designed with psychometric literature well-known characteristic of splitting the problem set into subgroups which has the same number of items. A mixed zero-one integer linear programming(MZOILP) are developed and solved in IBM CPLEX ILOG [31].

A heuristic is produced which only deals with the 28 data packet of the hard28 set by De la Rosa et al. [36] They compared the time to solve all 28 instances of hard28 with previous studies' timings on the same data set. Hard28 data set has two important characteristics; one of them is its' one third of items' weights are heavier than half the bin sizes and the other is about a six of its' weights are prime numbers.These characteristics cause large items in first bins get alone before applying bin-oriented heuristic like FFD. The heuristic are designed to deal with the characteristics of hard28 data set and reduce the time need to solve the instances.

2.2 Grouping Genetic Algorithms

Many optimization problems have been a well study area to find a better solution. On the other hand, in decades scientist looked at the environment and saw how living cells survived with adapting harsh conditions. With the recent development in the Biology and Genetics, a GA is developed as a method to find better solutions for many optimization problems using the same reproductive functions in gene level to build new generations. A GA typically starts with the generation of the initial population

which are randomly found solutions. Then, it eliminates the solutions which have undesirable characteristics. Finally reproduction occurs with crossover and mutation. The elimination process in each level is mentored by a fitness function which calculates the fitness of the solution. Nature of the evaluation, solutions, which have a better fitness value, are given more priority to be an offspring.

The GGA is a version of GA and it has been altered to cope with grouping problems. Falkenauer was the first one who implemented GGA in different problems. GGA has two characteristics different than standard GA. First a special encoding scheme is used to make the relevant structures of grouping problems become genes in chromosomes. The standard chromosome structure is changed and subgroups named group part is added. Second, given the encoding, special genetic operators are used, suitable for the chromosomes [18]. By solving the problem set of BPP and adding dominance criterion to the GGA, a hybrid grouping genetic algorithm(HGAA) for BPP showed that GGA hold promise both for bin packing and other grouping problems [19].

A heuristic named *better-fit*, which simply replace the item with remaining items if it fits better, is produced by Bhatia and Basu. Results of the study matches the quality of Falkenauer's HGGA, but in an efficient computational time differently [28].

Rohlfshagen and John A. Bullinaria presents a simple steady-state genetic algorithm developed using an evolutionary approach to solve hard optimization problems which conventional approaches tend to fail [45]. The algorithm provides an approach to solve the BPP without the need for any additional heuristics. The design of the algorithm presented here is an example of group encoding, developed by molecular genetics inspiration, which allows for a modularization of the search space in which individual sub-solutions may be assigned independent cost values. Following the modularization, crossover process modeled on the theory of exon shuffling is applied on to the values to select an offspring that inherits the most promising segments from its parents. Here, purpose of the crossover process is to ensure that only one offspring, which inherit the most tightly packed bins from both parents, is chosen and also it ensures that the bins with minimum slack are preserved for further consideration in order to find optimal number of bins. The crossover process in exon shuffling approach is greedy in terms of finding the best bin first. Three sets of online avail-

able hard BPP problem instances are utilized to test proposed algorithm. The results of runs show that algorithm is able to find global optimum solution for 8 out of 10 instances. The cases which optimal solution cannot be found, the solution is only one bin away from the optimal. Fitness (bin slack), number of bins, and number of constraint violations are studied to investigate overall behavior of the algorithm and it verified that this current biologically inspired exon shuffling GA has the highest rank in terms of success rate over the existing algorithms. The algorithm first orders the bins by slack space increasing. Then all the same level bins has a chance to be in the offspring. In the second phase of the algorithm deleting of the same bins occurs. For Scholl and Klein's hard10 data set, it found 8 optimal solution while MBS, MBS′, Perturbation MBS′, Sampling MBS′, FFD, BFD, B2F has 0, Relaxed MBS′ has 2, GA, BISON has 3 optimal solution for the same data set. The only exception is that Alvim et al.'s study found 10 instances as optimal [1].

An algorithm works by reducing the distance between slacks of the bins and item-weights distribution progressively on histograms, is presented by Poli et al. [42]. The algorithm has two main procedures; a procedure to choose the item to be inserted and a procedure to chose how to priorities gaps in order of urgency. These two procedures works progressively and reduce the gap on histogram.

Singh and Gupta introduced two different heuristics. One of them is hybrid steady-state grouping genetic algorithm (H-SGGA) and the other is improved version of Perturbation MBS′ which is based on the study of Fleszar and Hindi (2002) [7][51] . In addition to that a combined heuristics of H-SGGA and an improved Perturbation MBS′ tested and named C_BP. The improved Perturbation MBS′ performance is better than Perturbation MBS′ and C_BP performance are better than Alvim et al.'s algorithm (2002) [1].

A simple, non-specialized, non-hybridized and evolutionary based algorithm is presented by Stawowy [52]. The algorithm has effective size reduction and unique concept of separators movements during mutation. It gives somehow less satisfactory solutions than its fancy competitors. The main focus of the study is verifying an evolutionary method's applicability to the bin packing problem and as the result, yes it can be applied to the BPP very well.

To lower the computational time of genetic algorithms, a parallel-island based multi-objectivized memetic algorithm, which is widely used as a synergy of evolutionary or any population-based approach with separate individual learning or local improvement procedures for problem search, is designed by Segura et al. for two-dimensional BPP [50]. Mono-objective problems can be transferred to multi-objective ones with avoiding local optima. In addition to that island-based models provided benefits in terms of solution quality, and in terms of time saving.

A hybrid heuristic, which combines the effectiveness of island parallel algorithms and GGA, is proposed by Dokeroglu and Cosar [38]. The developed parallel hybrid grouping genetic algorithm make use of the multiprocessors to set up the several distinct population randomly and search for a better solution for a given number of generation. The algorithm has several steps; firstly it generates the population and truncate it to a pre-defined population size in each processor, than it add BFD and FFD to the population, after crossover and mutation procedures, items that couldn't fit for any bin are inserted as BFD or FFS or MBS. 1,318 benchmark sets are analyzed and 88.5% of the instances are solved with optimal solution . In addition to that, Parallel GGAs generates distinct sub-populations concurrently, thus probably assembling better solution quality by providing many chromosomes and generations.

An algorithm named as Grouping Genetic Algorithm with Controlled Gene Transmission (GGA-CGT) for BPP is described by Quiroz-Castellanos et. al. [20]. The algorithm chooses the best genes in the chromosomes for the offspring without losing the balance between the selective pressure and population distinction. New grouping genetic operators is used to convey the best genes. Even more the evolution is controlled by a new reproduction technique which eases finding the best genes and avoids incomplete convergence of the algorithm. The improved performance with choosing the best genes and binding and rearrangement heuristics can be seen in the results gathered from the study. GGA-CGT finds 16 optimal solutions of hard28's 28 instances while Alvim et al.'s study, Pert.-SAWMBS and BFD find only 5 optimal solution to the same 28 data instances [1].

2.3 Parallel Optimization with CUDA

The computational power of GPUs significantly outperforms present multi-core processors (CPUs), and so there is an increased interest in using graphics cards for solving combinatorial problems. One of the earliest study with a combinatorial problem is done by Janiak A. et. al. [40] . The traveling salesman and flow shop scheduling problems are tried to be solved with a parallel tabu search algorithm. General-Purpose Computation Using Graphics Hardware, which is the ancestor of the CUDA platform, are used and noted that GPU can be even 16 times raster than CPU implementation at that time.

An integer programming model,Knapsack Problem, are studied on a Tesla Platform which is a multi GPU computing system architecture by Boyer, V. et. al. [29]. Single CPU is connected to a multi GPU structure and a heterogeneous programing is implemented. The implemented design is referred as robust, because timing results are stayed approximately the same even with an increase in size of the problem.

A parallel island-based genetic algorithm which shows one of the best examples of the heterogeneous computing model described by Pospichal, P. et al. [43]. Up to our knowledge, there are not many works using homogeneous computing model for NP-hard combinatorial problems. The tests are conducted with Griewank, Michalewicz and Rosenbrock's benchmarks and the speed up is promising. The only drawback of the algorithm is transfer times between host and device. The design scheme of the algorithm can be seen in Figure 2.1 . In addition to that Built-in CUDA timer functions is used to get kernel execution times. The results also show that the proposed GPU implementation of GA can provide better results in the shorter time or produce better results in equal time.

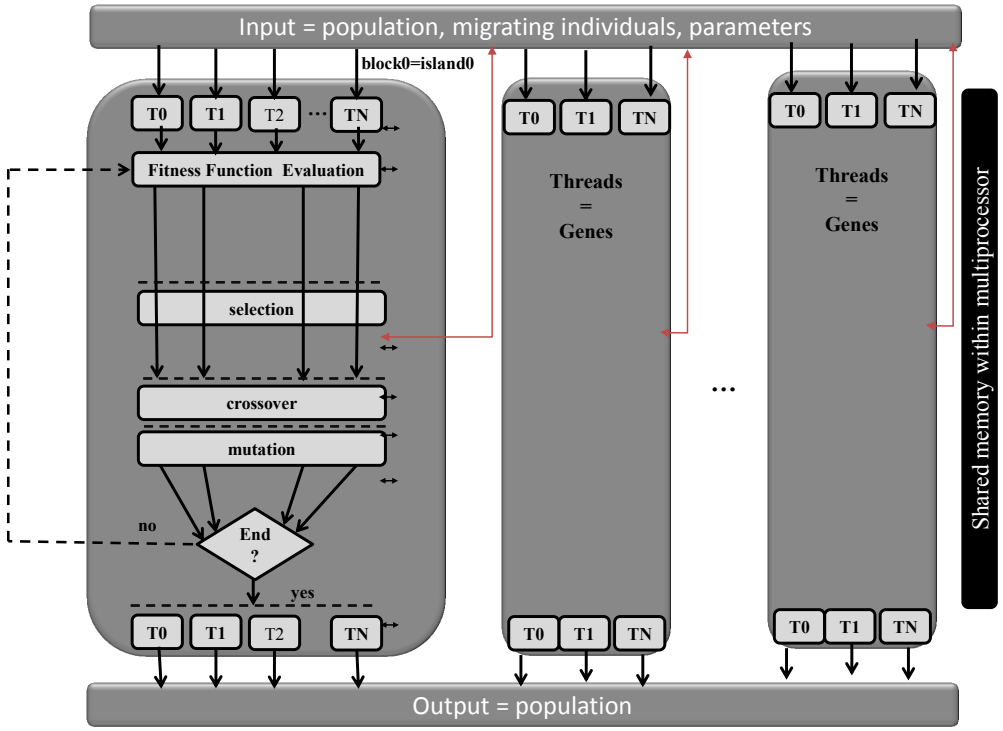

Figure 2.1: An Example of Parallel Genetic Algorithm

A cluster of GPU computing units named multi-GPU cluster is used in the study which presents distributed tabu search meta-heuristics [30] . The algorithm is based on another study which is about distributed parallelizations with cooperation. It is controlled by a master process to communicate between concurrent threads. The results are compared with 1,2 and 3 processors which have 11.66,10.03 and 6.37 speed up respectively.

A parallel multi-start tabu search (PMTS) for quadratic assignment problem is proposed by M. Czapiński(2012) [35] . Quadratic Assignment Problem, an NP-hard problem, is studied with detailed analysis of parallelization possibilities, memory organization and access pattern. The timing results of the PMTS are showed us it is 420x faster than single-core CPU implementation, while 70x faster than a parallel CPU implementation with a six-core CPU.

One of the other combinatorial optimization problem is the vehicle routing problem (VRP) . If the problem has 50 visiting points in a direct solution, there will be 30^{64} distinct solution. The inaccessibility to try out all possibilities caused a need for

high performance computing. Cekmez, U. et. al. presents an implementation of GA with 1-Thread in 1-Position (1T1P) approach[33] . The size of population is increased with maximizing efficiency to execute the algorithm parallel on GPU. Also as the other studies, GPU and CPU implementation is compared. As a result, GPU implementation has a speed up ratio varying 366 to 1955.

One of the other heuristic search algorithm named Gravitation Field Algorithm (GFA) is proposed by Rong et.al.[46] to be optimized with running parallel on CUDA platform. The algorithm is based on island model which minimize the need for communication between concurrent threads. It was compared with uni-modal version and noted that the parallel version is not only have speedup, but also accuracy. One of the instance of the six standard benchmark functions is took 319735 ms with GFA while 14179 ms with Parallel GFA.

A Parallel ant colony optimization(ACO) on graphics processing units is presented by Delévacq et.al [37] . It is an example of heterogeneous computing which performs tour construction on the GPU and pheromone update on the CPU. It performs 3-opt local search for solving the TSP. After the first construction of a block, a new an tour and the local search is implemented. Then, three edges from the tour is deleted with reassigning them to have a resulting tour shorter than the initial tour if this is at all possible. The algorithm is very complex and it takes a lot of time but can yield near optimal results. In addition to that the proposed study achieved the maximum speed-up factor of 23.9 over the CPU implementation.

The resource constrained project scheduling problem, a complex problem even for small instances, is studied by Bukata et.al. [32] . A parallel Tabu Search algorithm is altered to maintain quality solutions with speedup. For J90 Benchmark problem, it showed 10.5/42.7 faster ratio than it's CPU counterpart.

2.4 A Simple CUDA Example

In this section we would like give an example of massive computational power of CUDA platform. The example is named as inner product or dot product and it takes two vectors, then it multiplies the corresponding elements of these two vectors and

finally calculates the sum of the products. The following equation shows the mathematical description of a simple dot product.

$$(x_1, x_2, x_3, x_4) \times (y_1, y_2, y_3, y_4 = x_1y_1 + x_2y_2 + x_3y_3 + x_4y_4) \tag{2.1}$$

A detailed information is given below with the Figure 2.2. The code in the figure shows the kernel part of the dot product algorithm.

```
__global__ void dot( float *a, float *b, float *c ) {
        __shared__ float cache[threadsPerBlock];
        int tid = threadIdx.x + blockIdx.x * blockDim.x;
        int cacheIndex = threadIdx.x;
        float temp = 0;
        while (tid < N) {
                temp += a[tid] * b[tid];
                tid += blockDim.x * gridDim.x;
        }
        // set the cache values
        cache[cacheIndex] = temp;
        // synchronize threads in this block
        __syncthreads();
        // for reductions, threadsPerBlock must be a power of 2
        // because of the following code
        int i = blockDim.x/2;
        while (i != 0) {
                If (cacheIndex < i)
                cache[cacheIndex] += cache[cacheIndex + i];
                __syncthreads();
                i /= 2;
        }
        if (cacheIndex == 0)
        c[blockIdx.x] = cache[0];
}
```

Figure 2.2: Kernel Function of Dot Product(which is taken from the book [24])

As seen above we declared a shared memory which all threads in a block can read and write. Each thread's running sum is stored in this fast accessed memory. The size of this shared memory is as *threadsPerBlock* which is an enough space for threads to write their summations.

First while loop designates where a parallel product operation occurs. For example a thread which has a tid equals to "0" multply array a's first element with array b's first element while other threads do the same operation with different corresponding

elements of the arrays. After all threads writes the product of elements to the shared *cache* memory with *cacheIndex*, a thread synchronize operation must be executed to continue rest of the operations. All thread execute the same instruction until the line of syncthreads() called and they wait for the other threads to complete the instruction. After all threads finish their job, execution continues with the reduction part of the algorithm.

In the reduction while loop, each thread will add corresponding two elements in shared memory. It will continue for log2(threadsPerBlock) and at the end the first element has the summation of the products and we give the job of writing the solution to the global memory to *tid* "0".

We conducted some test to have a better understanding of CUDA platform. First we increased the array size to see how execution time changes, and the results presented in Table 2.1.

Table 2.1: Increasing Array Size

Size of the Array	Execution Time (sec)
$1\times1024\times1024$	2.57
$2\times1024\times1024$	4.64
$4\times1024\times1024$	8.11
$8\times1024\times1024$	17.00
$16\times1024\times1024$	33.58

Then we examined how *ThreadPerBlock* size might effect the execution times with the array size of $128\times1024\times1024$

The results are presented in Table 2.2 . "512" *ThreadPerBlock* size have the optimal configuration for the algorithm.

Table 2.2: Increasing ThreadPerBlock Size

ThreadPerBlock	Execution Time (ms)
1	43.96
2	25.78
4	16.75
8	12.28
16	10.23
32	8.97
64	8.31
128	**7.96**
256	8.03
512	8.11
1024	8.19

listings

CHAPTER 3

PROPOSED ALGORITHM

As discussed in previous chapters, we tried to solve one dimensional bin packing problem with an aim to have the same results in shorter time or getting better results in the same time than its CPU implementations. Through our study we tried to learn as much as we can from GPU computations and GAs. A standard GA can be seen in Algorithm 1 .

Algorithm 1 A Standard Genetic Algorithm

1: **function** GA(*sizeOfpopulation*)
2: Generate random population of n Chromosomes
3: Evaluate the fitness $f(x)$ of each chromosome x in the population
4: $n \leftarrow$sizeOfpopulation.length
5: **for** $i \leftarrow 1, n$ **do**
6: [Selection] Select two parent chromosomes from a population according to their fitness (the better fitness, the bigger chance to be selected)
7: [Crossover] With a crossover probability cross over the parents to form a new offspring (children). If no crossover was performed, offspring is an exact copy of parents.
8: [Mutation] With a mutation probability mutate new offspring at each locus (position in Chromosome).
9: [Accepting] Place new offspring in a new population
10: Use new generated population for a further run of algorithm
11: pack rect(i) into new bin
12: **end for**
13: **end function**

3.1 Chromosome Structure

Based on two main reasons, Falkenaur's chromosome structure is chosen for our study. GGAs nature is the primary reason that drive our decision since GGAs work better with special encoding scheme of Falkenaur's chromosome structure.[44, 18, 19]. Additionally, Holland-style chromosome structure has drawbacks such as invalid offspring generation and degeneration.

3.2 Exon Shuffling Crossover Operator

The selection procedure is variable to study. r chromosomes are picked from the solution population and the one which has the highest value of interest are passes to next generation from these previously chosen r elements. In this study we preferred to select items randomly. After selection of the parents, an offspring is generated using different crossover techniques. A two phase exon shuffling crossover technique is utilized to generate offspring of our study. Items of each parent are grouped in bins and then the bins of parents are combined together. In next step, the bins are ordered according to their fitness value. Minimum slack criterion is used as the fitness value of our study. Remaining item list is another crucial part of exon shuffling crossover. If items of the current optimal bin are exist in the remaining item list, then it can be added to the offspring's chromosome as a new gene. If not, next bin of the ordered list is evaluated using the same approach.

We used exon shuffling crossover [45, 12], a recent technique borrowed from molecular genetics, for our proposed parallel algorithms. A Field of biology and genetics, called molecular genetics, deals with genes in molecular level and mimics the molecular methods and interactions among genes. After a two phase crossover, an offspring is generated. First phase takes two chromosomes and chooses the genes with best qualities to pass them to the offspring, while the second phase reinsert the remaining items.

3.3 Mutation Operator

The mutation operator enables new solutions using the current optimal solution. In this study, the mutation operator works based on the population size and predefined mutation ratio. Number of groups chosen change depending on the population size and mutation ratio. The mutation operator works on number of groups computed as multiplication of population size and mutation ratio and select that number of groups randomly. The items of the selected groups are removed from the current solution list and they are added to remaining item list. At then end of mutation process, items in the remaining item list are inserted back to groups in the solution list using BFD algorithm. However, mutation operator does not guarantee a solution with less number of bins than the current optimal solution. If slack of the existing groups are not appropriate for the items in remaining item list, then the algorithm can add new groups to the remaining items.

3.4 Inversion Operator

Inversion operator is applied to increase the transfer probability of fitter gene pair to the next generation. At the beginning of process, two groups are chosen simultaneously from each chromosome and their slack and groups numbers are interchanged. The upcoming crossover and mutation operators take place on these interchanged sets. The inversion operator provides an increased opportunity for promising future generations without changing the item list during the operation.

3.5 Fitness function

A fitness value, which will allow us to choose most promising chromosome, is computed based on an equation defined by Falkenauer :

$$FF = \sum_{i=1}^{N} (\frac{F_i}{c})^k \tag{3.1}$$

There are different approaches to compute a fitness value in order to lead choice procedure. Some of the approaches to calculate fitness value increase the solution space by keeping suboptimal solutions. From the other side if we only prefer to use group size as the fitness value, better solutions can be discarded. As a result, the choice of fitness function (FF) requires additional caution.

In Equation 3, N is the number of bins, F_i is the total weights of the items packed into the gene i ($i = 1$,..., N), c is the bin size, and k is a heuristic exponential number. The value k prefers the most filled genes rather than the ones has plenty of space. Study [18] uses $k = 2$ but the study [52] proposed that $k = 4$ demonstrates higher qualified solutions.

3.6 Proposed Algorithm, CUDA-GGA-1DBPP

Our algorithm for BPP consist of following main functions;

- Setting up Host and Device Allocations
- Generation of Arrays with Random Numbers - on GPU asynchronously
- Generation of Initial Population - on GPU asynchronously
- Truncate Population to Population Size - on CPU
- Add BFD solution to the Population - on CPU
- Find slacks of each bin in every chromosome - on GPU asynchronously
- Crossover - on GPU asynchronously
- Calculate Fitness Value for each solution - on GPU asynchronously
- Mutation - on GPU asynchronously
- Calculate Fitness Value for each solution - on GPU asynchronously
- Validate and display results - on CPU

The pseudo code of our proposed algorithm;

Algorithm 2 Our Proposed Algorithm for 1D-BPP

1: Setting up Host and Device Allocations
2: Generation of Arrays with Random Numbers ($sizeOfpopulation$*$TRatio$)
3: Transfer generated population from *Device* to *Host*
4: Truncate Population and add Best Fit Decreasing Solutions to the population ($sizeOfpopulation$)
5: Transfer truncated population from *Host* to *Device*
6: Find slacks ($numberOfGroups$)
7: Evaluate the fitness $f(x)$ of each chromosome x in the population
8: **function** GENERATION($sizeOfpopulation$)
9: $n \leftarrow$numberOfgeneration
10: **for** $i \leftarrow 1, n$ **do**
11: Crossover ($sizeOfpopulation * CrossoverRatio$)
12: Mutation ($sizeOfpopulation$)
13: Place fittest offsprings into population
14: Inversion ($sizeOfpopulation$)
15: **end for**
16: **end function**
17: Validate and display results

A general schema of device and host functions in time-line can be seen in Figure 3.1

.

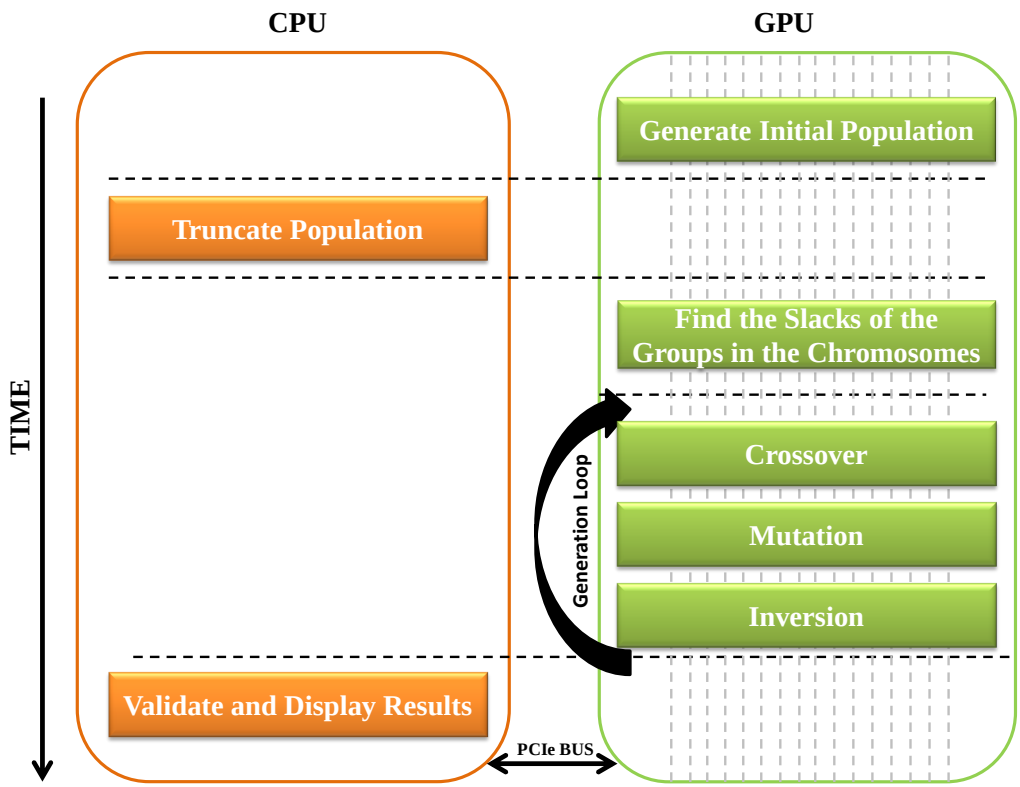

Figure 3.1: Algorithm Schema

3.6.1 Setting up Host and Device Memory Allocation

Since it is an GA we need enough memory space on both device and host to generate population freely. So First of all we allocate memory on host side with the following command;

```
chromosome  population = (chromosome )calloc(sizeofpopulation
    Tratio, sizeof( chromosome)) ;
```

chromosome is a structure which has following variables in it;

```
int groupnamesofthe items[500];
int groups[500];
int groupsize;
int slackofthe group[500];
float fitnessvalue FF;
```

group_names_of_the_items is an array which describes the group names of each items. For example if first item which is [0]. element in the array has a bin where it is deployed, the value of the [0] is the bin's number or group number in GGA. Groups array stores the group numbers and group_size is the solution that shows how many bin/group is used to pack the problem data set items.

Slack_of_the_group is another array which shows the residual capacity of the groups and float fitness_valueFF store the solution's Fitness Value. Since we can't use vectors, we need to determine the size of the array with the maximum number of items.

The reason why the size of allocation is *T_Ratio* times grater than size of population is to generate initial population in order to truncate to the size of population with best ones of the population. *cudaMalloc* allocate memory on GPU with the *g_population* device pointer and *T_ratio* times greater than the size of population

In addition to that, since we need a memory space to generate initial population, we allocated exactly the same sized space in the device;

*cudaMalloc((void**)g_population, tratio*size_of_population* sizeof(chromosome))*

3.6.2 Generating Random Numbers on GPU

For mutation and generation of initial population we need to have random numbers to select from wide selection possibilities. Since CUDA platform is not have standard libraries of C languages, generation of random numbers seem to little problematic. One way to have them is, firstly generate and append to an array, then copy the array to the GPU side, the other option is to use CURAND library of GPU side. We prefer second option, not only for we need it on GPU side but also for having the specialty to generate them parallel.

A basic generation of CURAND is used in our study. We generate two different random set in order to use it sequentially. To generate the numbers CURAND needs a state information;

```
curandState   devState;
curandStateMRG32k3a   devMRGStates;
```

We send the state pointers to kernels to make the states ready for the generation-kernels. In this study we used two different generation states to have completely different two 1000-element arrays. One of them generated by MTGP32 pseudorandom sequence generator which is an NVIDIA's adaption of an algorithm proposed by Saito et.al [47] . The other state we used is CURAND's default state which generates an array of pseudorandom numbers greater than 2^{190} .

In addition to what we have done above for generation, "Kernel Concurrency" and "Host-Device Memory Copy Concurrency" are used to do asynchronous operation for generation of two distinct random numbered arrays. Three streams are created totally in this step. First two of three is used for the generation, and the last one used for asynchronous memory copy of item weights from host to device. These three operations are completely independent and can run asynchronously. There is only one exception in the generation which is the generation hugely depends on setting up kernel states. So these two kernels must run synchronously.

3.6.3 Generation of Initial Population

As a standard GA an initial population is generated with the random numbered arrays which gathered in previous section. After allocation enough memory on the device the kernel which executes the generation procedure, is luanched with the following configuration;

Block Size: $(N + fill\_groups\_ThreadSize - 1)/fill\_groups\_ThreadSize$

Thread Size: $fill\_groups\_ThreadSize$

As seen above, a variable named *N* corresponds to product of *population_size* with *T_Ratio* and helps to calculate the amount of thread of block size. Block size is tailored to generate enough thread for pre-specified and tunable *fill_groups_ThreadSize* . Size of population and *fill_groups_ThreadSize* are given in MetaData file read initially even before the reading of data set. The size of population and number of generation are two tunable variables to examine if the better results can be acquired with increasing them. With the above formulation the kernel grantees to have block and thread sizes to produce *T Ratio* times *Population Size* concurrent GPU processes.For exam-

ple if a population size is defined as "350" and T ratio is defined as "30"the kernel will be executed with concurrent 10500 processes to generate initial population.

After generating each chromosome, the population array is filled with the chromosomes to the " population[tid].th "element resident on the device memory.

3.6.4 Truncate Population

A population of chromosomes is generated in previous step. Next consequent step is truncating population to the size of population with selecting best chromosomes of the population. Eliminating the chromosomes which have solutions not as well as the others means better generations or offspring especially for mutation and crossover operators.

It is a simple host function and it does not have any heavy part to run on GPU asynchronously. So we decided not to run it on GPU. The only drawback in this step is the population which reside in device memory need to be copied to the host.

3.6.5 Reinserting Items with BFD Algorithm

As described in Chapter 1&2 BFD is one of the uncomplicated heuristic algorithms for solving the BPP. The algorithm's pseudo code can be seen as following;

Algorithm 3 Best Fit Decreasing (BFD) algorithm

```
1:  function BFDH(itemsList, binCapacity)
2:      sort itemsList by decreasing height
3:      n ←itemsList.size
4:      for i ← 1, n do
5:          minRemainingSpace ← binCapacity
6:          m ←bins.size
7:          for j ← 1, m do
8:              if item(i) fits into bin(j) then
9:                  pack item(i) into bin(j)
10:                 if remainingSpace < minRemaningSpace then
11:                     minRemainingSpace ← remainingSpace
12:                 end if
13:             end if
14:         end for
15:         if item(i) fits fit into more than one bin then
16:             repacks the item into minRemainingSpace bin level
17:         else
18:             create new bin
19:             pack item(i) into new bin
20:         end if
21:     end for
22: end function
```

3.6.6 Finding Slacks of the Groups

Crossover and Mutation operators need the last status information of the groups to rearrange them. The one of the crucial information of the groups is how much slacks do we have to put new items. At beginning the the study we had two opportunities for calculating the slacks on the device. The first option is to running the each chromosome concurrently and the second option is to running the chromosome sequentially while in each step calculating the slacks of the each bin concurrently. We chose the

latter one because it utilizes the device more than the first one for even increasing the population size. So each chromosome of the population goes in the algorithm, then its groups divided into parallel processes to calculate its' slacks. After finishing the last chromosome, we have a population which all chromosomes have the information about amount of residual capacity of each its' groups.

3.6.7 Crossover

While designing the algorithm, we implemented the crossover operator on the device firstly. But running the algorithm for two randomly chosen chromosomes concurrently caused us to get TDR (Timeout Detection Recovery) from operating system which is a protection to have a responsive GPU. The reason why we get TDR is the crossover's complex structure. As we discussed in chapter 1, GPU core is quite different from a CPU core. GPU core is designed for highly parallel lightweight operations while a CPU core is designed for complex and single processed operations. This difference is pushed us to run the operator on host side . For example, hard28 data set is took *50-70* seconds with executing the crossover on device while max *7* seconds on host side (For total data set and complete algorithm) . So we decided to run the Crossover on the host initially.

Then we searched the problem and found out the structure of the remaining item has the main responsibility. Searching for a remaining item whether it is deleted or not is an exhausting procedure. So we decided to put a deleted *Boolean* to the structure of the remaining item list. In this way, the complexity of the algorithm is decreased and with other minor changes we could perform the crossover function on GPU.

In addition to than we divided the function to three main kernels. One of them is to prepare remaining items and offspring for the crossover, second one is the crossover and the final one is to pack remaining items to the generated offspring.

After we have a courage to implement the operator on the GPU, we implemented three different crossover operators but finally we can run it with exon shuffling method which chooses the best groups from the chromosomes with an aim to have better offspring. Also we added a variable named "Crossover Ratio" which defines the size

of the offspring population after multiplication with population size.

3.6.8 Mutation on GPU

Two main parts of a general GA are Crossover and Mutation. For Mutation we successfully implemented it on device. Mutation is less complex than crossover. Firstly every block takes a distinct chromosome. Then every block chooses a group via random numbers reside on the device. This group's items are deleted from the each chromosome. After reinserting the items deleted with BFD, we have a chance to have better slacks on the array which have the slacks of the groups. Rather than calculating slacks of every group, we preferred to update the slack information every time when an item inserted or deleted. In this way we pushes all blocks to execute a mutation in itself without communicating other processes.

Another constant variable we read from the meta data text file, is number of generation. The algorithm executes the Crossover and Mutation for a definite number of times.

To increase the chance for fitter chromosomes we also added the "Mutation Ratio" to our proposed algorithm. A for loop which runs for *"population size * mutation ratio"* times. Rather than deleting just a group, more than one number of groups are deleted and inserted to the remaining item list. The solution quality is increased with added mutation ratio.

3.6.9 Calculating Fitness Values

For calculating the fitness values of each chromosome, we preferred to have an enough block size division of size of population by 64 and 64 threads. We likely have better performance results than run it by blocks which have one thread per. So every chromosome's Fitness Values are calculated by concurrent blocks and threads.

As we discussed, communication between host and device has a price. Since item weights are constant values, it doesn't need to transfer back from device to host. But the population is needed to transfer from the device to host after the initial generation

on the device for the truncating and adding BFD to the population. After these functions we need to transfer the population back again to the device to find slacks. For crossover,mutation and calculating fitness values, the population is transferred to the device again. Finally after the last function in the last generation on GPU, we transfer it back to host for validating and displaying the results. At that time we no longer need the Random Numbered Arrays, item values and population on the device. So the final operation takes place on the device is to free the memory they are occupied on GPU.

CHAPTER 4

EXPERIMENTAL RESULTS

In this section, we briefly explain our environment, then give information about problem data sets, configuration settings and results for the data sets, finally conclude with general analyzes of the gathered results.

4.1 Experimental Environment

Two different personal computers are used for the experiments. One for CUDA implementation tests and the other is CPU implementation to compare the results. The main PC, which has a CUDA capable graphical processing unit, has following hardware capabilities; For CPU implementations;

- Intel Core i5-2467M CPU 1.60 GHz with 4 cores
- 4 GB Memory (RAM)
- Windows 7 64bit Operating System

is used.And for GPU implementations;

- Intel Pentium 3258 CPU 3.20 GHz with 2 cores
- 8 GB Memory (RAM)
- Windows 7 64bit Operating System
- EVGA NVIDIA GeForce GTX 750 Ti Graphical Processing Unit

is used. Our GPU, named as NVIDIA GeForce GTX 750 Ti, is a mid-sized GPU designed for both gaming and computing environment 4.1:

Figure 4.1: GPU Card [27]

Main specifications of the GPU can be seen in Table 4.1.

4.2 Problem Data Sets

In our study we used three sets of problem instances for the experiments . The instances are set_1, set_2 , set_3 [48] and hard28 [54]. The benchmark problem data sets are presented in Table 4.2.

4.3 Configuration Settings For The Proposed Algorithm

As we described in previous chapters, launching a kernel with "N" *Blocks* contains one *Thread* in each, equals to launching with one *Block* contains "N" *Thread* in terms of generating N software depended parallel processes. But execution times can be different for each configuration.Since time is not luxury we have, we tried to select best block and thread sizes to read minimum execution times.

Table 4.1: GPU's Technical Specifications

Specifications	**Values**
Architecture Implementation	7
Architecture Revision	162
Number of GPCs	1
Number of TPCs	5
Number of SMs	5
Warps per SM	64
Lanes per warp	32
Register file size	65536
Max CTAs per SM	32
Max size of shared memory per CTA (bytes)	49152
SM Revision	327680
Driver	WDDM
GPU Family	GM107-A
MAX THREADS PER BLOCK	1024
MAX BLOCK DIM X	1024
MAX BLOCK DIM Y	1024
MAX BLOCK DIM Z	64
MAX GRID DIM X	2147483647
MAX GRID DIM Y	65535
MAX GRID DIM Z	65535
MAX SHARED MEMORY PER BLOCK	49152
TOTAL CONSTANT MEMORY	65536
WARP SIZE	32
MAX REGISTERS PER BLOCK	65536
MULTIPROCESSOR COUNT	5
KERNEL EXEC TIMEOUT	0
INTEGRATED	0
MEMORY CLOCK RATE	2700000
GLOBAL MEMORY BUS WIDTH	128
L2 CACHE SIZE	2097152
MAX THREADS PER MULTIPROCESSOR	2048
MAX SHARED MEMORY PER MULTIPROCESSOR	65536
MAX REGISTERS PER MULTIPROCESSOR	65536
COMPUTE CAPABILITY MAJOR	5
COMPUTE CAPABILITY MINOR	0
TOTAL MEMORY	2147483648

Table 4.2: The benchmark problem data sets

problem instance	**# instances**	**item weights**	**bin capacity (*c*)**	# items (*n*)
set_1	720	[1,100]	{100, 120, 150}	{50, 100, 200, 500}
set_2	480	[3, 9] items at each bin	1,000	{50, 100, 200, 500}
set_3	10	[20,000, 35,000]	100,000	200
hard28	28	[1, 800]	1,000	{160, 180, 200}

For each data set we tried to find best "Block and Thread Sizes" to get minimum execution times before conducting the tests. Since Set_1 and set_2 have relatively more number of instances, the configuration setting tests are studied for only 50 instances of them. How the setting effects the execution time is presented in Table 4.3 for Set_1 data set, Table 4.4 for Set_2 data set, Table 4.5 for Set_3 data set and Table 4.6 for hard28 data set. Gray shaded configurations are chosen for the tests. For example, as seen in Table 4.4, settings for *Fill Groups Randomly* function which is used for generation of initial population with *28-BlockSize* and *64-ThreadSize* and *Generation* stands for the functions which needs a total process number equals to population size with *5-BlockSize* and *16-ThreadSize* has best execution time for 40 *Number of Generations* and 80 *Population Size*. Other tests which force to tune these parameters are conducted with saving the ratio between Block and Thread Sizes. Set_2 data set in Table 4.4, and Set_3 data set in Table 4.5 have the most attractive examples of how the Block and Thread Size configuration can effect execution time. The best configurations have approximately 40 seconds while the worst ones have 60 seconds execution time for 50 instances.

Table 4.3: Configuration Settings for Set_1 Data Set(720 Problem Instances)

Function	#Blocks /Grid	#Threads /Block	Time(Sec) for 1st 50 Problem Instances
Fill Groups Randomly	14	128	23.09
Generations	3	32	
Fill Groups Randomly	28	64	17.82
Generations	5	16	
Fill Groups Randomly	56	32	19.51
Generations	10	8	
Fill Groups Randomly	112	16	17.95
Generations	20	4	
Fill Groups Randomly	224	8	18.68
Generations	40	2	
Fill Groups Randomly	448	4	20.44
Generations	80	1	

Table 4.4: Configuration Settings for Set_2 Data Set(480 Problem Instances)

Functions	#Blocks /Grid	#Threads /Block	Time(Sec) for 1st 50 Problem Instances
Fill Groups Randomly	8	256	60.01
Generations	3	32	
Fill Groups Randomly	16	128	53.54
Generations	5	16	
Fill Groups Randomly	32	64	47.44
Generations	10	8	
Fill Groups Randomly	64	32	51.27
Generations	20	4	
Fill Groups Randomly	128	16	47.50
Generations	40	2	
Fill Groups Randomly	256	4	38.58
Generations	80	1	

Table 4.5: Configuration Settings for Set_3 Data Set(10 Problem Instances)

Function	#Blocks /Grid	#Threads /Block	Time(Sec)
Fill Groups Randomly	14	128	61.28
Generations	3	32	
Fill Groups Randomly	28	64	51.29
Generations	5	16	
Fill Groups Randomly	56	32	47.24
Generations	10	8	
Fill Groups Randomly	112	16	44.70
Generations	20	4	
Fill Groups Randomly	224	8	40.08
Generations	40	2	
Fill Groups Randomly	448	4	39.09
Generations	80	1	

Table 4.6: Configuration Settings for hard28 Data Set(28 Problem Instances)

Function	#Blocks /Grid	#Threads /Block	Time(Sec)
Fill Groups Randomly	14	128	54.55
Generations	3	32	
Fill Groups Randomly	28	64	49.91
Generations	5	16	
Fill Groups Randomly	56	32	46.80
Generations	10	8	
Fill Groups Randomly	112	16	47.02
Generations	20	4	
Fill Groups Randomly	224	8	46.28
Generations	40	2	
Fill Groups Randomly	448	4	49.65
Generations	80	1	

4.4 Settings The Population Size

We tried to increase the *Population Size* with a predefined and constant *Number of Generations* and *Truncate, Crossover, Mutation, Inversion Ratios*. The Results for the Set_1 data set are presented in Table 4.7 and the charts as "Population Size vs. Number of Optimal Solutions" can be seen in Figure 4.2, "Population Size vs. Total Number of Extra Bins" in Figure 4.3 and "Population Size vs. Execution Time" in Figure 4.4. *Number of Optimal Solutions* shows the amount of optimal solution with comparing every instance with given optimal solutions for each data set instances.*Total Number of Extra Bins* shows the summation of extra bins which is calculated by subtracting found best solution, which is group/bin number required to pack all items, with the best solution for each data set instances.

Table 4.7: The Effect of Changing Population Size for Set_1 Data Set(720 Problem Instances)

Set_1 with 720 Problem Instances	
Number of Generations	40
Random Population Initialization Product	20
Crossover Ratio (% of population)	50%
Mutation Ratio	20%
Inversion Ratio	20%

Population Size	# of Optimal Solutions	# of Extra Bins	Time (Sec)
20	574	212	1239.00
40	584	174	1374.57
60	612	117	1570.78
80	622	102	1696.64
100	614	108	1701.86
150	613	110	2233.97
300	611	124	3712.35

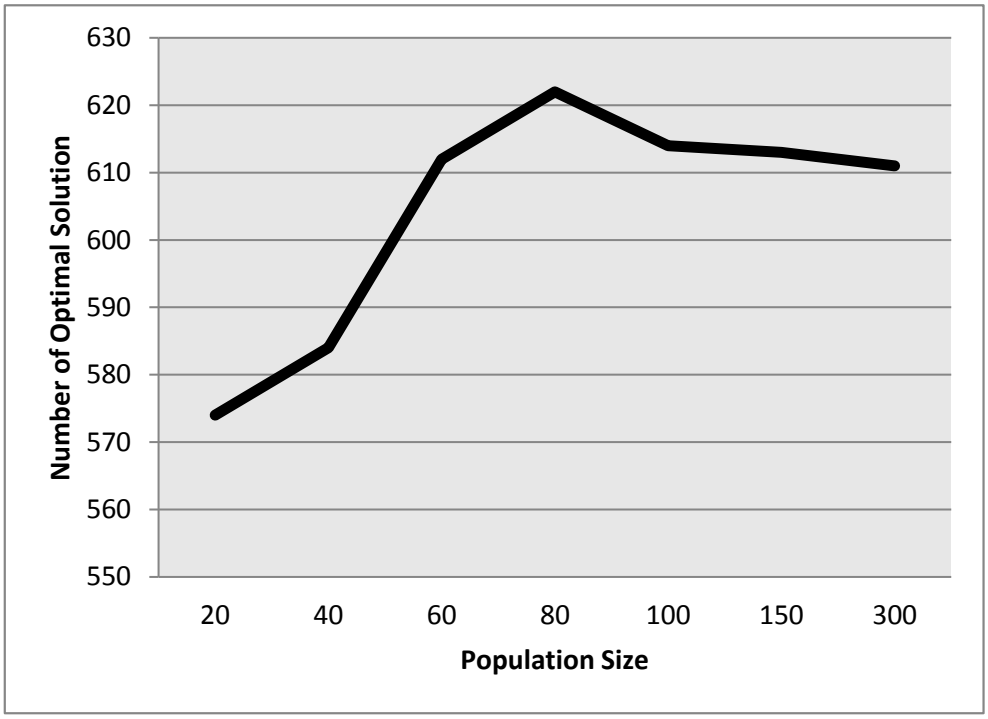

Figure 4.2: Population Size vs. Number of Optimal Solution for Set_1(720 Problem Instances)

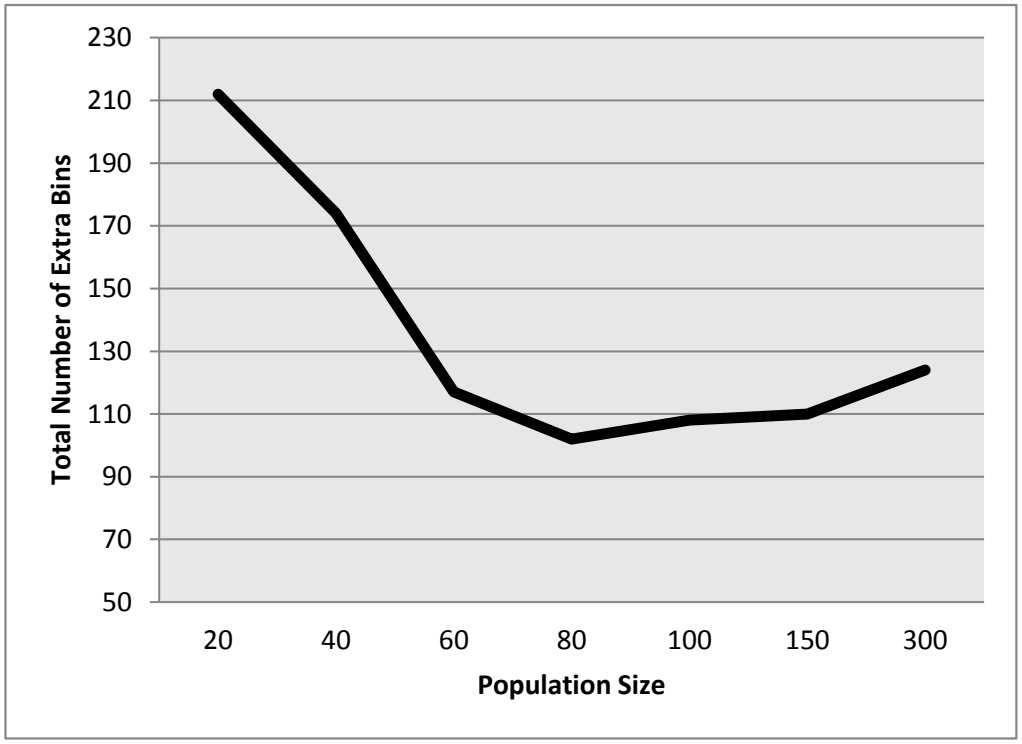

Figure 4.3: Population Size vs. Total Number of Extra Bins for Set_1(720 Problem Instances)

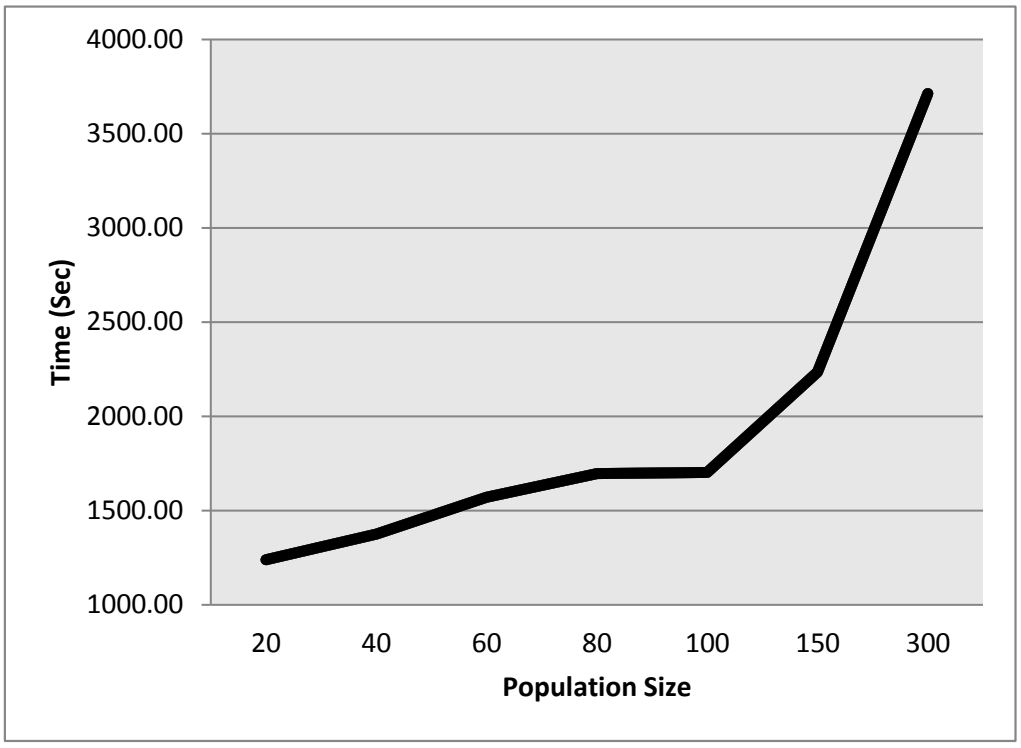

Figure 4.4: Population Size vs. Execution Time for Set_1(720 Problem Instances)

In our first design, we only change population size and run the algorithm for the predetermined fixed values of the configuration parameters on Set_1 data set. When we look at the Figure4.3 and Figure 4.4, it is obvious that there is a negative correlation between the number of optimal solutions and total number of extra bins required as expected. However, there is a change of trend for both parameters starting from the population size 80. We conclude that increase in population size has a limited effect on number of optimal solutions when number of generations is constant. That is why we observe a decrease as population size increases 80 through 300. Execution time, another parameter of interest, follows an increasing trend. The time required to solve algorithm displays a steepest slope when population size is in between 20 and 100 then when population size is in between 100 and 300. Global memory constraint may be the result of this steepest slope.

Following the Set_1 data set, we run the algorithm for Set_2 data set using the same configuration by only changing the population size. The correlation between the number of optimal solutions and total number of extra bins and the relation between the population size and execution time has a similar trend with the results of Set_1 data set. The results for the Set_2 data set are presented in Table 4.8 and the charts as "Population Size vs. Number of Optimal Solutions" can be seen in Figure 4.5, "Population Size vs. Total Number of Extra Bins" in Figure 4.6 and "Population Size vs. Execution Time" in Figure 4.7.

Table 4.8: The Effect of Changing Population Size for Set_2 Data Set(480 Problem Instances)

Number of Generations	40
Random Population Initialization Product	20
Crossover Ratio (% of population)	50%
Mutation Ratio	20%
Inversion Ratio	20%

Population Size	# of Optimal Solutions	# of Extra Bins	Time (Sec)
20	299	292	730.32
40	338	196	755.37
60	378	140	770.54
80	381	126	885.88
100	372	128	999.98
150	373	139	1387.99
300	372	138	2485.03

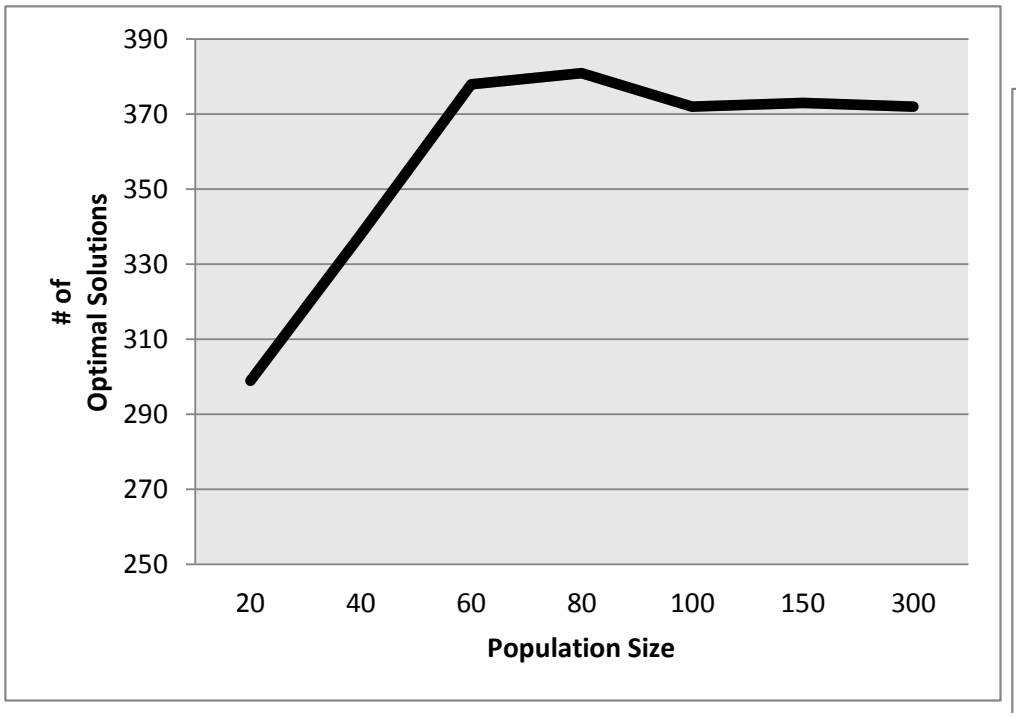

Figure 4.5: Population Size vs. Number of Optimal Solution for Set_2(480 Problem Instances)

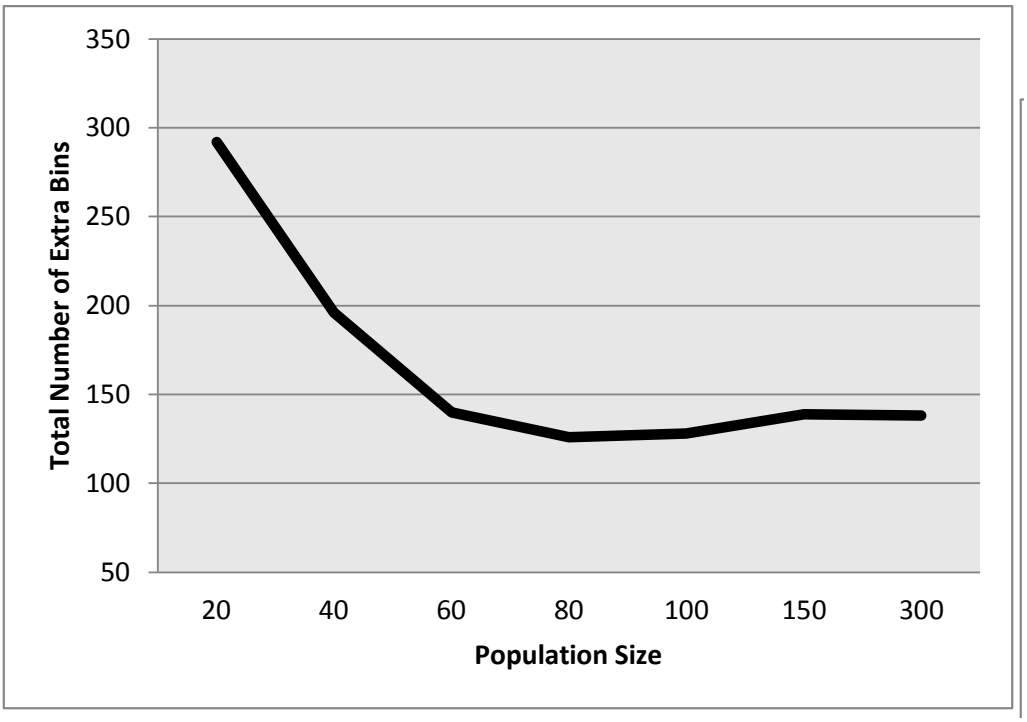

Figure 4.6: Population Size vs. Total Number of Extra Bins for Set_2(480 Problem Instances)

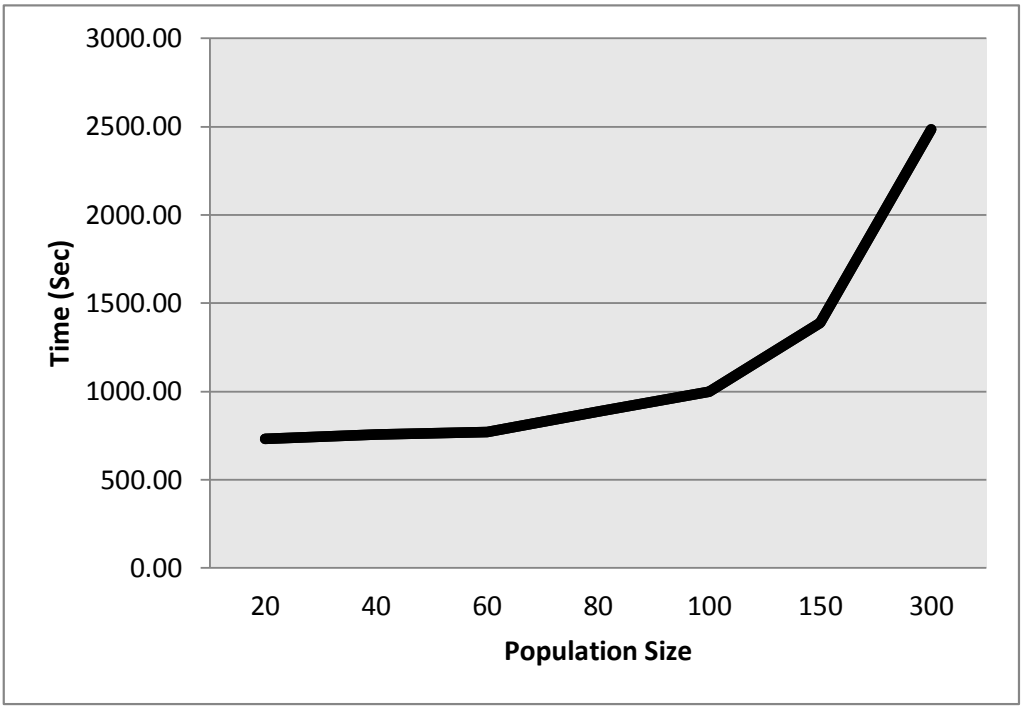

Figure 4.7: Population Size vs. Execution Time for Set_2(480 Problem Instances)

Suggested algorithm cannot find an optimal solution for Set_3 data set for same configuration and execution time follows an increasing trend as before. The results for the Set_3 data set are presented in Table 4.9 and the charts as "Population Size vs. Total Number of Extra Bins" is presented in Figure 4.8 and "Population Size vs. Execution Time" in Figure 4.9. Since we couldn't find any optimal solution for this data set we didn't present any chart related with it.

Table 4.9: The Effect of Changing Population Size for Set_3 Data Set(10 Problem Instances)

Number of Generations	40
Random Population Initialization Product	20
Crossover Ratio (% of population)	50%
Mutation Ratio	20%
Inversion Ratio	20%

Population Size	# of Optimal Solutions	# of Extra Bins	Time (Sec)
20	0	76	25.85
40	0	88	30.70
60	0	102	36.23
80	0	108	39.09
100	0	116	47.83
150	0	110	123.46
300	0	109	164.38

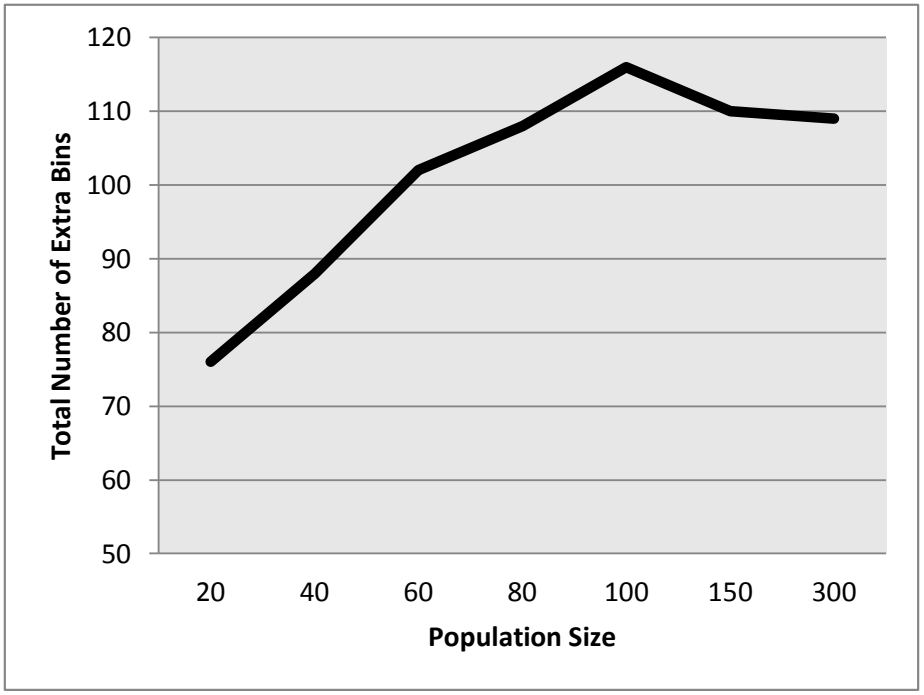

Figure 4.8: Population Size vs. Total Number of Extra Bins for Set_3(10 Problem Instances)

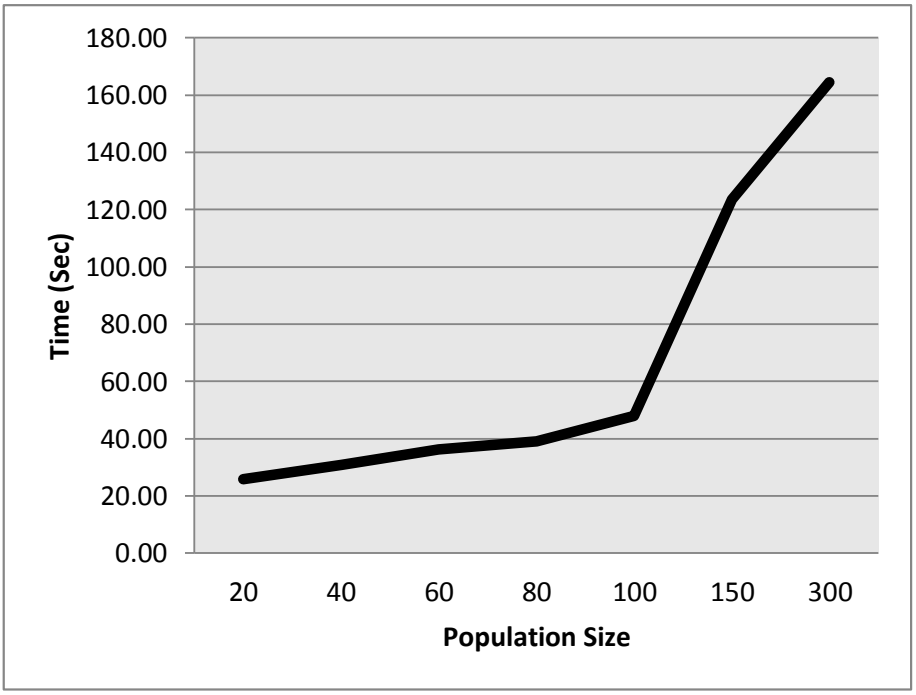

Figure 4.9: Population Size vs. Execution Time for Set_3(10 Problem Instances)

The results for the hard28 data set are presented in Table 4.10 and the chart of "Population Size vs. Execution Time" in Figure 4.10.Our algorithm ends up with an optimal solution of 5 for the hard28 data set. The execution time increase with the population size and execution time still has a drastic slope after the population size 100. Since the number of optimal solutions and total number of extra bins are not changed for this data set during the configuration change we didn't present any chart related with them.

Table 4.10: The Effect of Changing Population Size for hard28 Data Set(28 Problem Instances)

Number of Generations	40
Random Population Initialization Product	20
Crossover Ratio (% of population)	50%
Mutation Ratio	20%
Inversion Ratio	20%

Population Size	# of Optimal Solutions	# of Extra Bins	Time (Sec)
20	5	23	31.18
40	5	23	33.61
60	5	23	38.73
80	5	23	46.35
100	5	23	42.26
150	5	23	57.04
300	5	23	98.02

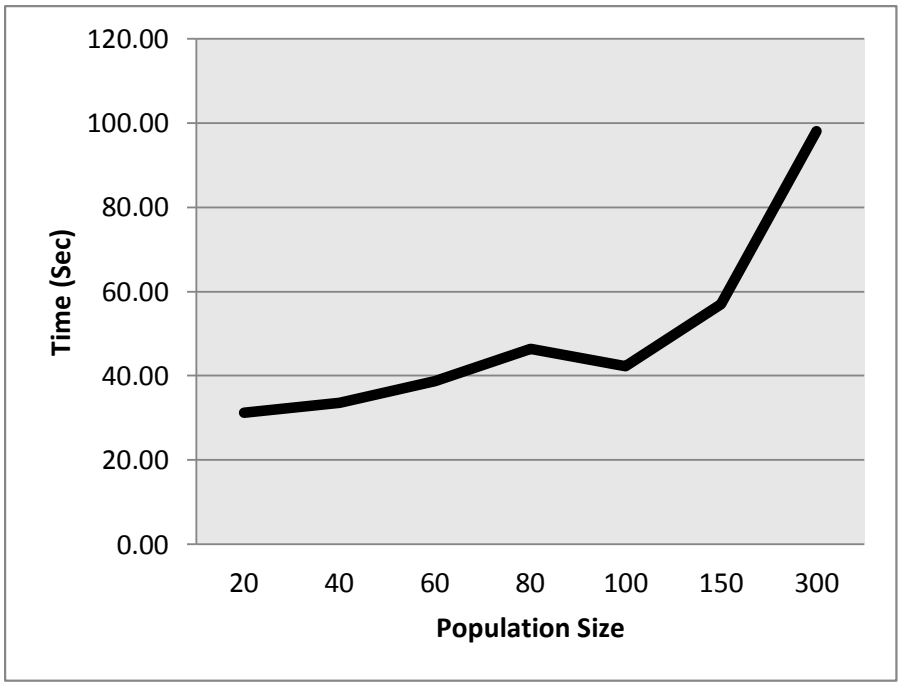

Figure 4.10: Population Size vs. Execution Time for hard28(28 Problem Instances)

4.5 Settings The Number of Generations

After finding best population size, we increased the number of generation to see how it effects the solution quality and execution times. As we discussed in previous chapters the *Number of Generation* corresponds to the number which crossover, mutation and inversion operations occurs repeatedly. When we run the algorithm for this given set up on Set_1 data set, number of optimal solutions stays as 619 after the number of generation 40 and so total number of extra bins required stays unchanged as expected. Additionally, execution time increases with the number of generations. The results for the Set_1 data set with each *Number of Generations*between "20" and "300" are presented in Table 4.11 and the chart as "Population Size vs. Execution Time" is shown in Figure 4.11 . Since there is no change in the *Number of Optimal Solutions* found and *Total Number of Extra Bins*, we didn't present any chart about them.

Table 4.11: The Effect of Changing the Number of Generations for Set_1 Data Set(720 Problem Instances)

Set_1 with 720 Problem Instances

Population Size	80
Random Population Initialization Product	20
Crossover Ratio (% of population)	50%
Mutation Ratio	20%
Inversion Ratio	20%

# of Generations	# of Optimal Solutions	# of Extra Bins	Time (Sec)
20	611	118	1038.01
40	619	107	1282.00
60	619	107	1457.57
80	619	107	1832.35
100	619	107	2205.55
150	619	107	3150.46
300	619	107	6171.10

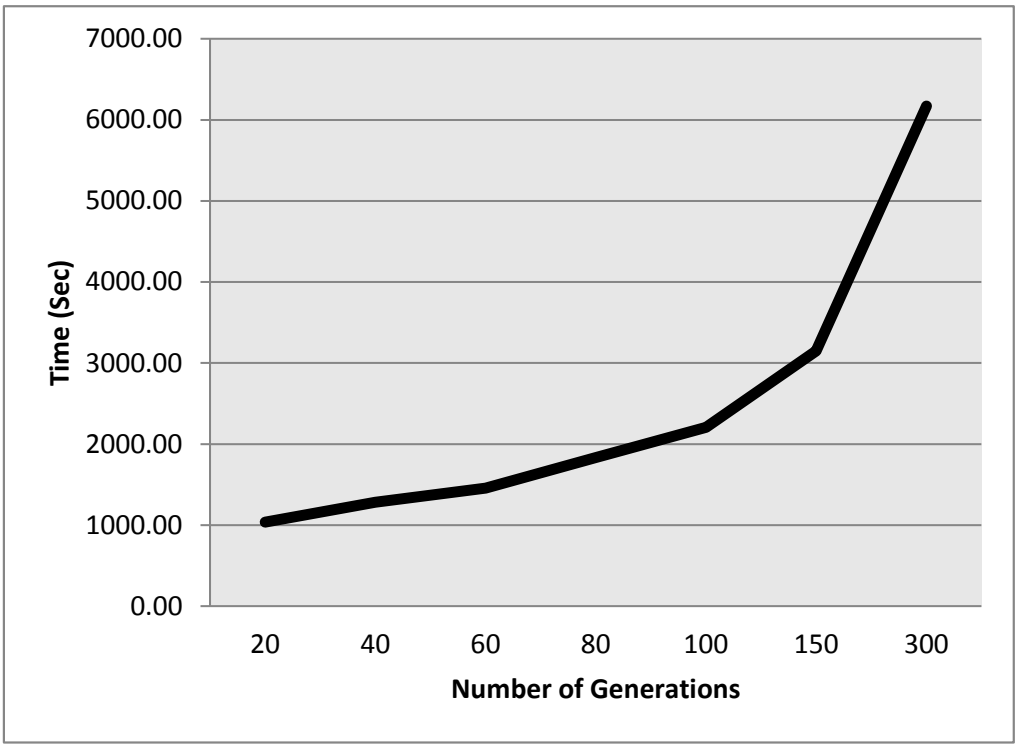

Figure 4.11: The Number of Generations vs. Execution Time for Set_1(720 Problem Instances)

When we run the same configuration for Set_2 data set, number of optimal solutions does not show a significant increase after the number of generation 40. Hence, total number of extra bins required follows an almost unchanged trend after the number of generation 40. Execution time increases with the number of generation as observed in previous tests. The results for the Set_2 data set are presented in Table 4.12 and the charts as "Number of Generations vs. Number of Optimal Solution" can be seen in Figure 4.12, "Number of Generations vs. Total Number of Extra Bins" in Figure 4.13 and "Number of Generations vs. Execution Time" in Figure 4.14.

Table 4.12: The Effect of Changing Number of Generations for Set_2 Data Set(480 Problem Instances)

Population Size	80
Random Population Initialization Product	20
Crossover Ratio (% of population)	50%
Mutation Ratio	20%
Inversion Ratio	20%

# of Generations	# of Optimal Solutions	# of Extra Bins	Time (Sec)
20	351	209	837.96
40	380	131	950.85
60	384	118	1288.54
80	384	117	1510.43
100	385	115	1933.06
150	385	115	3463.36
300	385	114	4642.20

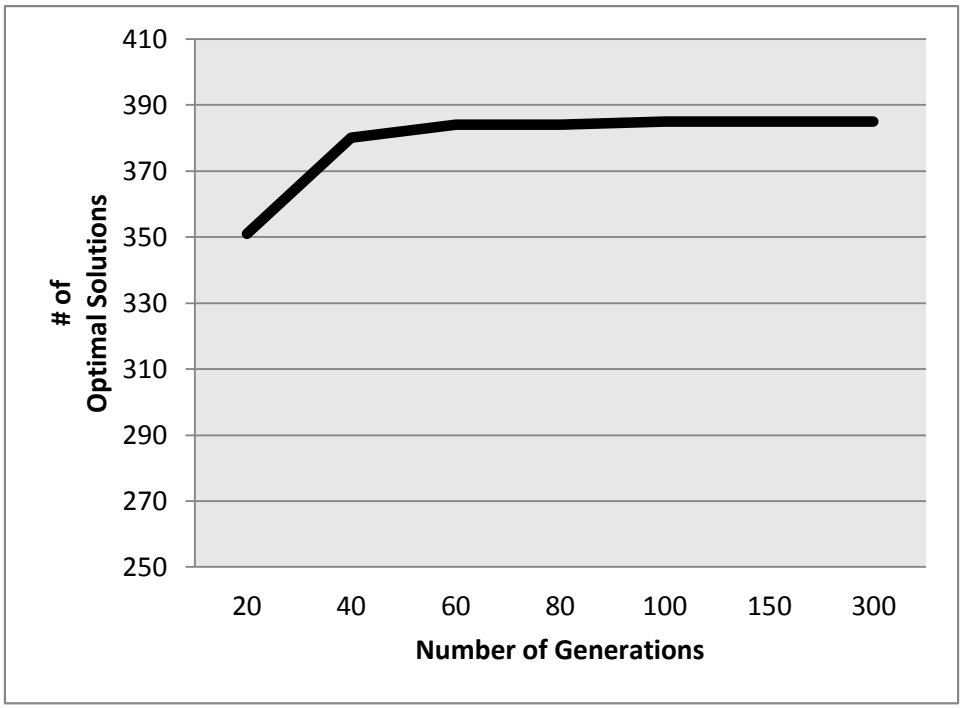

Figure 4.12: Number of Generations vs. Number of Optimal Solution for Set_2(480 Problem Instances)

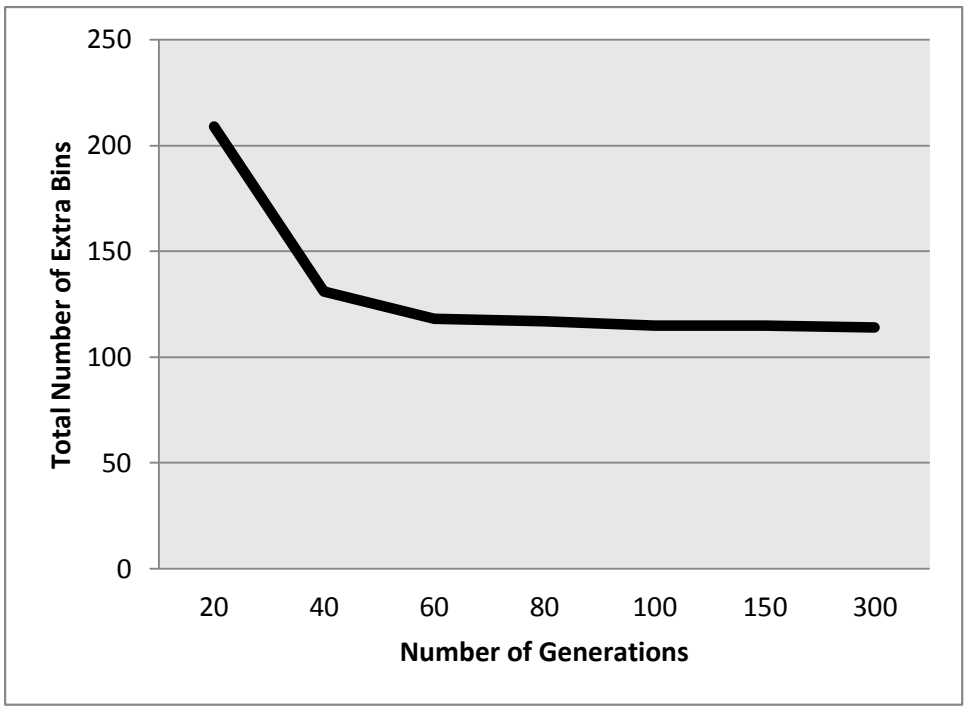

Figure 4.13: Number of Generations vs. Total Number of Extra Bins for Set_2(480 Problem Instances)

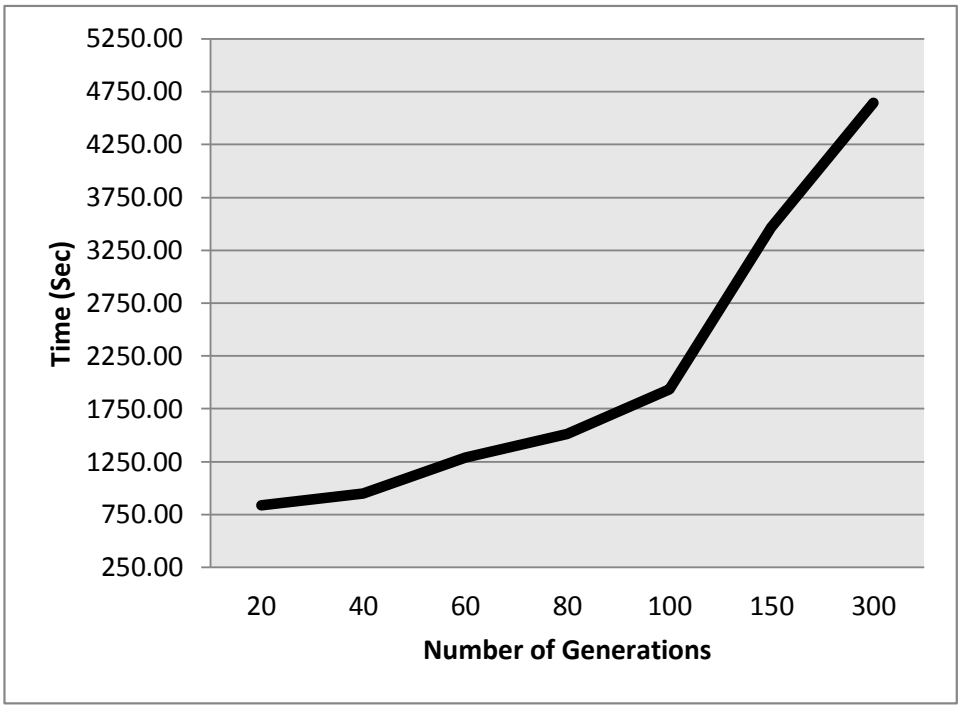

Figure 4.14: Number of Generations vs. Execution Time for Set_2(480 Problem Instances)

Our proposed algorithm cannot find an optimal solution for the Set_3 data set for our current configuration. By the way, execution time follows an increasing trend with number of generations.The results for the Set_3 data set are shown in Table 4.13 and the charts as "Number of Generations vs. Total Number of Extra Bins" in Figure 4.15 and "Number of Generations vs. Execution Time" are presented in Figure 4.16 .

Table 4.13: The Effect of Changing Number of Generations for Set_3 Data Set(10 Problem Instances)

Population Size	80
Random Population Initialization Product	20
Crossover Ratio (% of population)	50%
Mutation Ratio	20%
Inversion Ratio	20%

# of Generations	# of Optimal Solutions	# of Extra Bins	Time (Sec)
20	0	61	22.57
40	0	72	40.36
60	0	72	56.25
80	0	72	72.18
100	0	72	87.89
150	0	72	125.00
300	0	72	242.89

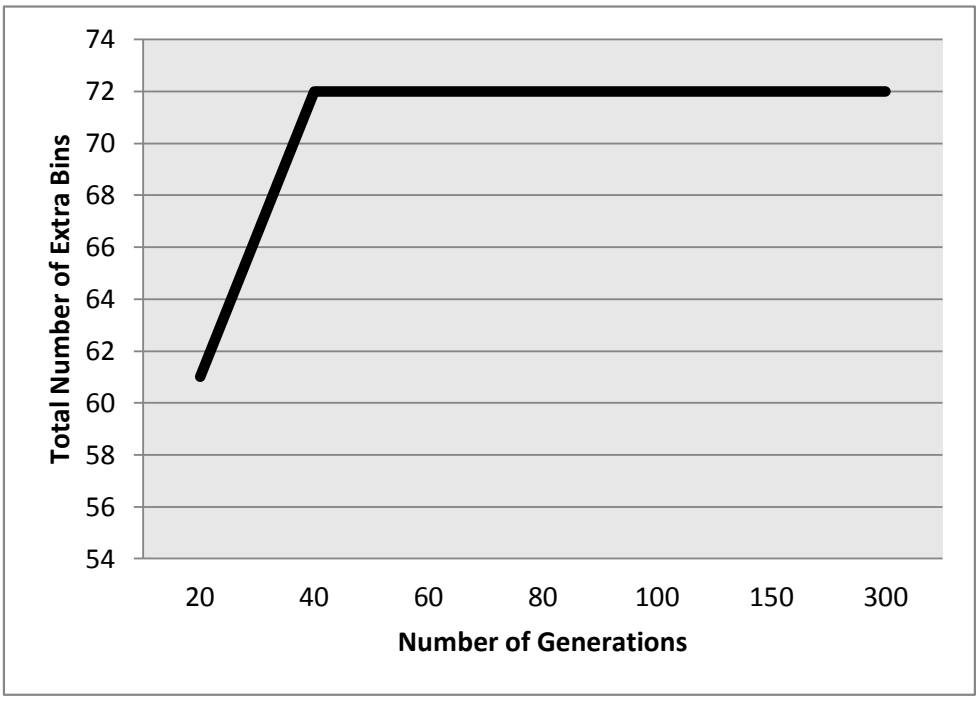

Figure 4.15: Number of Generations vs. Total Number of Extra Bins for Set_3(10 Problem Instances)

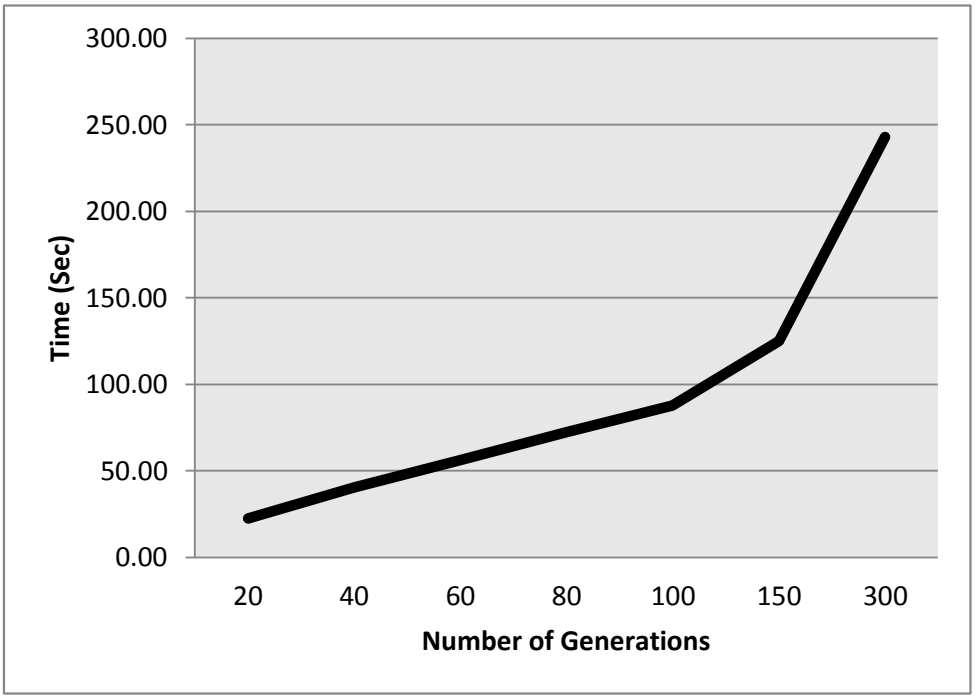

Figure 4.16: Number of Generations vs. Execution Time for Set_3(10 Problem Instances)

The number of optimal solutions turns out to be 5 for the hard 28 data set and number of extra bind required is determined to be 23. Trend in execution time is similar with the previous runs. The results for the hard28 data set are presented in Table 4.14 and the chart of "Population Size vs. Execution Time" in Figure 4.17.

Table 4.14: The Effect of Changing Number of Generations for hard28 Data Set(28 Problem Instances)

Population Size	80
Random Population Initialization Product	20
Crossover Ratio (% of population)	50%
Mutation Ratio	20%
Inversion Ratio	20%

# of Generations	# of Optimal Solutions	# of Extra Bins	Time (Sec)
20	5	23	30.36
40	5	23	46.24
60	5	23	61.21
80	5	23	76.19
100	5	23	73.05
150	5	23	93.72
300	5	23	240.94

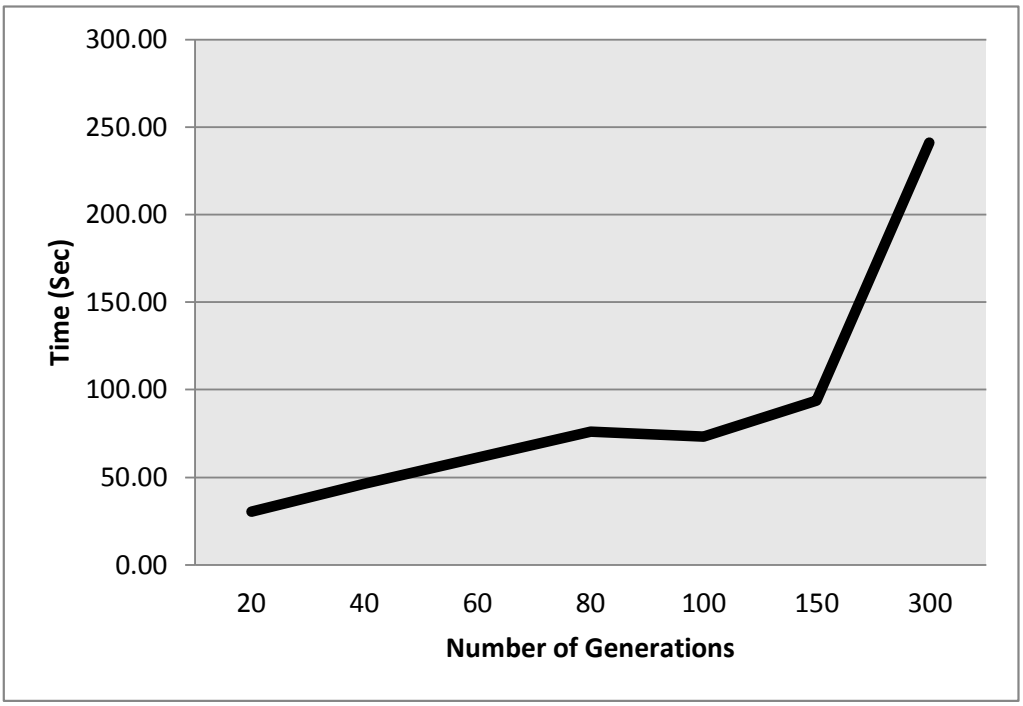

Figure 4.17: Number of Generations vs. Execution Time for hard28(28 Problem Instances)

4.6 Settings Random Population Initialization Product

Random Population Initialization Product is a production factor for population size to have a bigger initial search space which consist of randomly generated chromosomes. We examined the effect of increasing the *Random Population Initialization Product* to the urge of finding more optimal solutions.

The number of optimal solutions displays a mild decrease after the *Random Population Initialization Product* 20 for Set_1 data set. Increasing only *Random Population Initialization Product* when number of generations is constant can be thought as increasing the population size again number of generations is constant. That is why we see a mild decrease with increasing *Random Population Initialization Product*. However, increasing the *Random Population Initialization Product* does not have a significant effect on number of optimal solutions when run on Set_1 data set. The proposed algorithm cannot find an optimal solution for Set_3 data set for given con-

figuration. An optimal number of solutions 5 and number of extra bins 23 are found for hard28 data set and execution time has an increasing pattern as *Random Population Initialization Product* increases.

Results for the Set_1 data set with each *Random Population Initialization Product* between "10" and "50" are presented in Table 4.15 and the chart as "Random Population Initialization Product vs. Number of Optimal Solutions" is shown in Figure 4.18. We preferred to fix *Number of Generations* as "40" and *Population Size* as "80" since they yield to have best number of optimal solutions in our tests.

Table 4.15: The Effect of Changing the Random Population Initialization Product for Set_1 Data Set(720 Problem Instances)

Set_1 with 720 Problem Instances	
Number of Generations	40
Population Size	80
Random Population Initialization Product	20
Crossover Ratio (% of population)	50%
Mutation Ratio	20%
Inversion Ratio	20%

Random Population Initialization (N x population)	# of Optimal Solutions	# of Extra Bins	Time (Sec)
10	622	100	1045.11
20	626	97	1089.27
30	619	104	1198.33
40	617	105	1254.07
50	619	108	1396.05

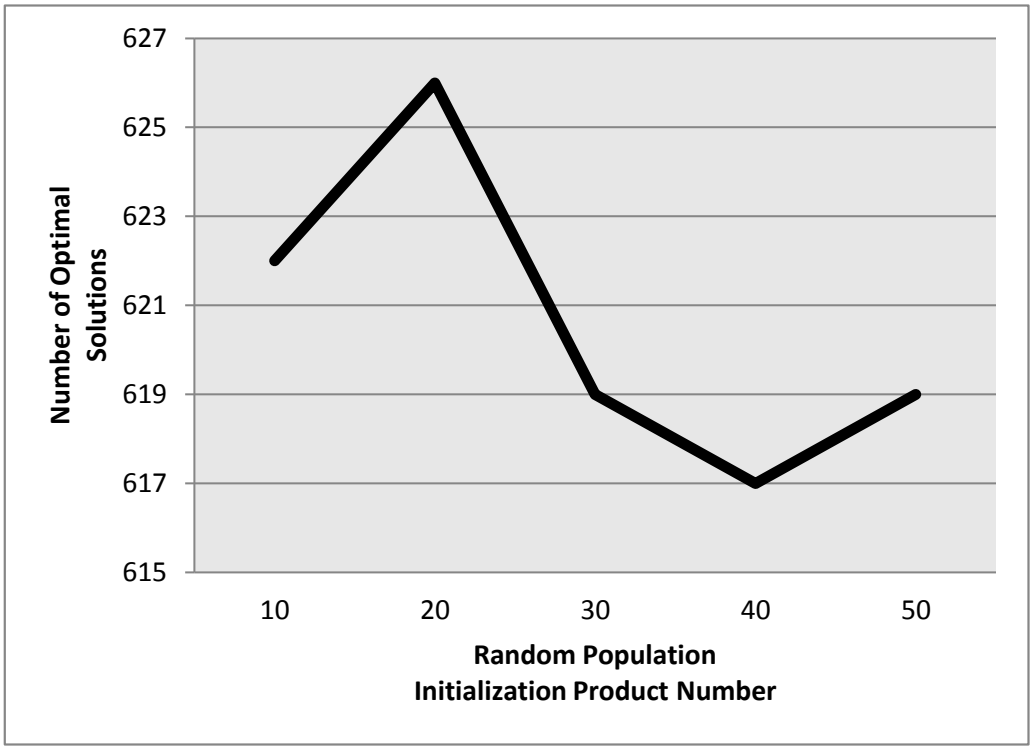

Figure 4.18: Random Population Initialization Product vs. Number of Optimal Solutions Set_1(720 Problem Instances)

The results for the Set_2 data set are presented in Table 4.16 and the chart of "Random Population Initialization Product vs. Number of Optimal Solutions" can be seen in Figure 4.19.

Table 4.16: The Effect of Changing the Random Population Initialization Product for Set_2 Data Set(480 Problem Instances)

Number of Generations	60
Population Size	80
Random Population Initialization Product	20
Crossover Ratio (% of population)	50%
Mutation Ratio	20%
Inversion Ratio	20%

Random Population Initialization (N x population)	# of Optimal Solutions	# of Extra Bins	Time (Sec)
10	382	115	1205.82
20	384	118	1288.54
30	380	122	1270.50
40	380	119	1296.42
50	379	118	1334.11

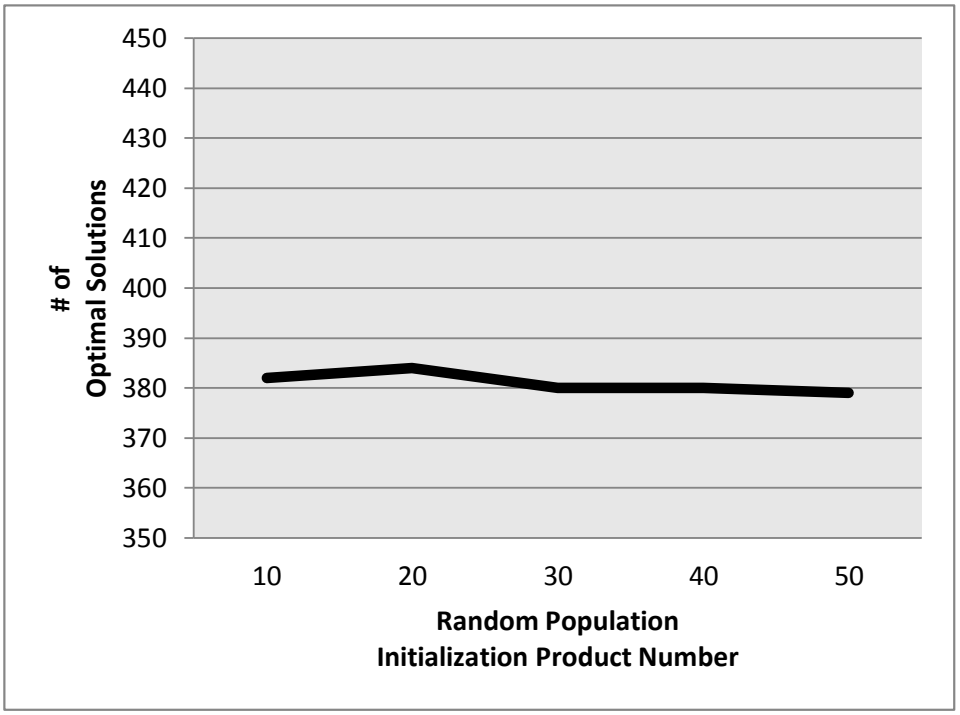

Figure 4.19: Random Population Initialization Product vs. Number of Optimal Solutions Set_2(480 Problem Instances)

The results for the Set_3 data set are presented in Table 4.17 and the chart of "Random Population Initialization Product vs. Total Number of Extra Bins" can be seen in Figure 4.20.

Table 4.17: The Effect of Changing the Random Population Initialization Product for Set_3 Data Set(10 Problem Instances)

Number of Generations	40
Population Size	80
Random Population Initialization Product	20
Crossover Ratio (% of population)	50%
Mutation Ratio	20%
Inversion Ratio	20%

Random Population Initialization (N x population)	# of Optimal Solutions	# of Extra Bins	Time (Sec)
10	0	114	45.44
20	0	72	40.41
30	0	68	40.10
40	0	70	41.93
50	0	63	45.64

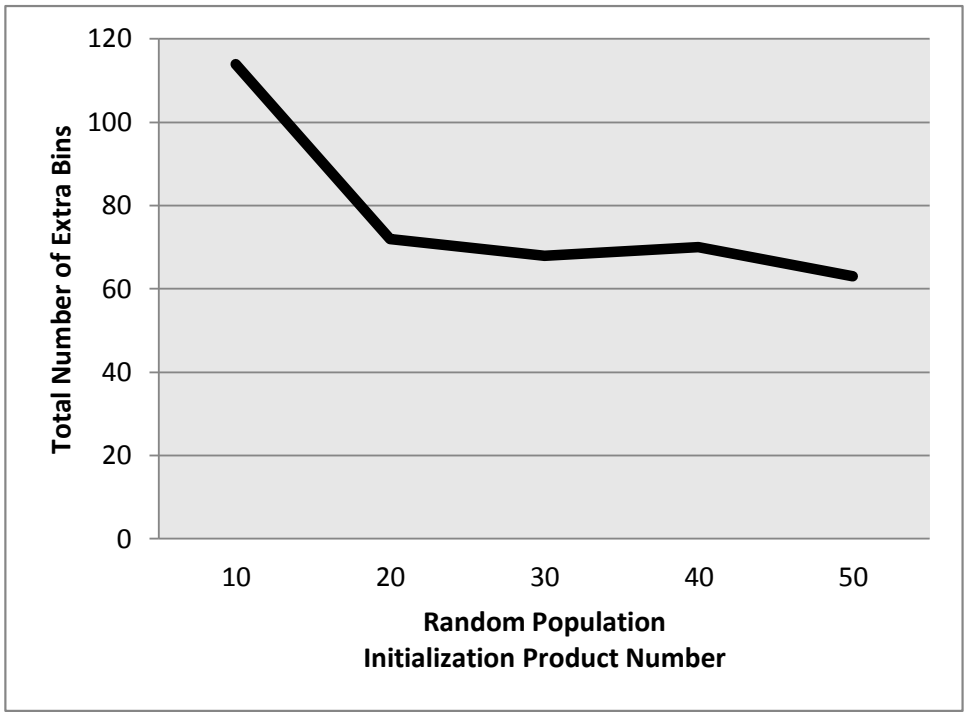

Figure 4.20: Random Population Initialization Product vs. Total Number of Extra Bins Set_3(10 Problem Instances)

The results for the hard28 data set are presented in Table 4.18 and the chart of "Random Population Initialization Product vs. Execution Time" in Figure 4.21.

Table 4.18: The Effect of Changing the Random Population Initialization Product for hard28 Data Set(28 Problem Instances)

Number of Generations	40
Population Size	80
Random Population Initialization Product	20
Crossover Ratio (% of population)	50%
Mutation Ratio	20%
Inversion Ratio	20%

Random Population Initialization (N x population)	# of Optimal Solutions	# of Extra Bins	Time (Sec)
10	5	23	45.72
20	5	23	46.23
30	5	23	46.30
40	5	23	48.41
50	5	23	48.42

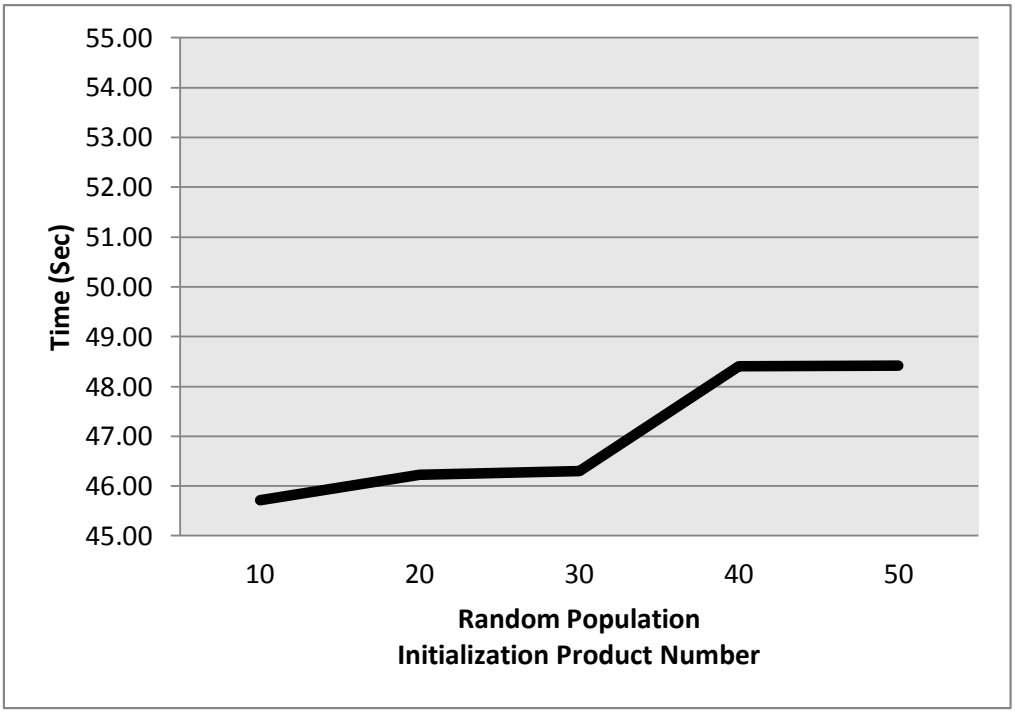

Figure 4.21: Random Population Initialization Product vs. Execution Time for hard28(28 Problem Instances)

4.7 Settings the Crossover, Mutation and Inversion Ratios

These three ratios are used in main operators of a GA. *Crossover Ratio* defines the offspring size which will be generated after the operation, while *Mutation and Inversion Ratios* corresponds to the size of array which will be generated in a mutation and inversion functions. We tried to select best effective ratios to find optimal or near optimal solutions.

Number of optimal solutions has an increasing pattern for Set_1 and Set_2 data sets. However, our proposed algorithm cannot find an optimal solution for Set_3 data set. Additionally, an optimal number of solution 5 and extra number of bins 23 are found as the result for hard28 data set.

For the mutation & inversion ratio, number of optimal solutions increase until the 636 when mutation & inversion ratio is 0,4. After the optimal level, number of optimal solutions starts decreasing for Set_1 data set. For Set_2 data set, increasing mutation

& inversion ratio does not have a significant effect on number of optimal solution. The proposed algorithm cannot find an optimal solution for Set_3 data set for different values of mutation & inversion ratio. The increase on mutation & inversion ratio does not change the results for hard28 data set. An optimal number of solution 5 and extra number of bins 23 are found for given configuration. For these two final configurations, designed by changing crossover and mutation & inversion ratio one at a time, execution time follows an increasing trend with increasing values of interest as observed in earlier experiments.

The results for the Set_1 data set with each *Crossover Ratio* between "0,2" and "0,8" and *Mutation&Inversion Ratios* between "0,1" and "0,5" are presented in Table 4.19 and the chart as "Crossover Ratio vs. Number of Optimal Solutions" is shown in Figure 4.22 and "Mutation&Inversion Ratio vs. Number of Optimal Solutions" is shown in 4.23 .

Table 4.19: The Effect of Changing the Crossover and Mutation&Inversion Ratio for Set_1 Data Set(720 Problem Instances)

Crossover Ratio (% of Population)	# of Optimal Solutions	# of Extra Bins	Time (Sec)
20%	609	127	993.43
40%	623	99	1062.45
50%	622	102	1696.64
60%	629	93	1728.41
80%	639	82	1423.70

Mutation&Inversion Ratio	# of Optimal Solutions	# of Extra Bins	Time (Sec)
10%	618	110	1056.84
20%	629	93	1150.56
30%	635	86	1275.73
40%	636	85	1202.81
50%	629	92	1249.60

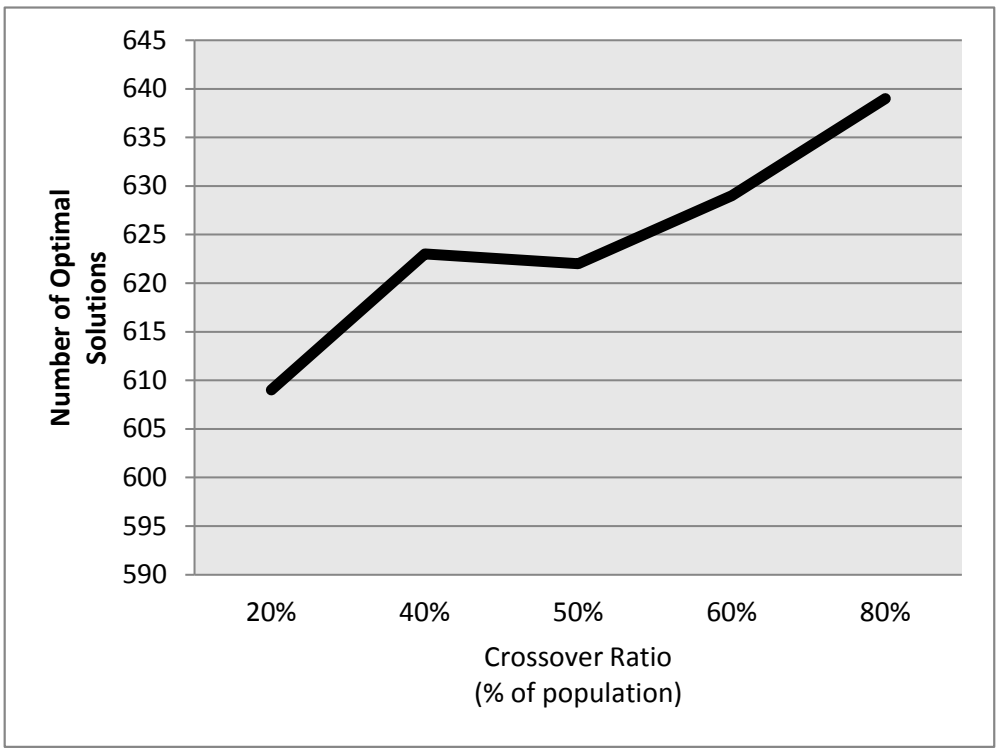

Figure 4.22: Crossover Ratio vs. Number of Optimal Solutions Set_1(720 Problem Instances)

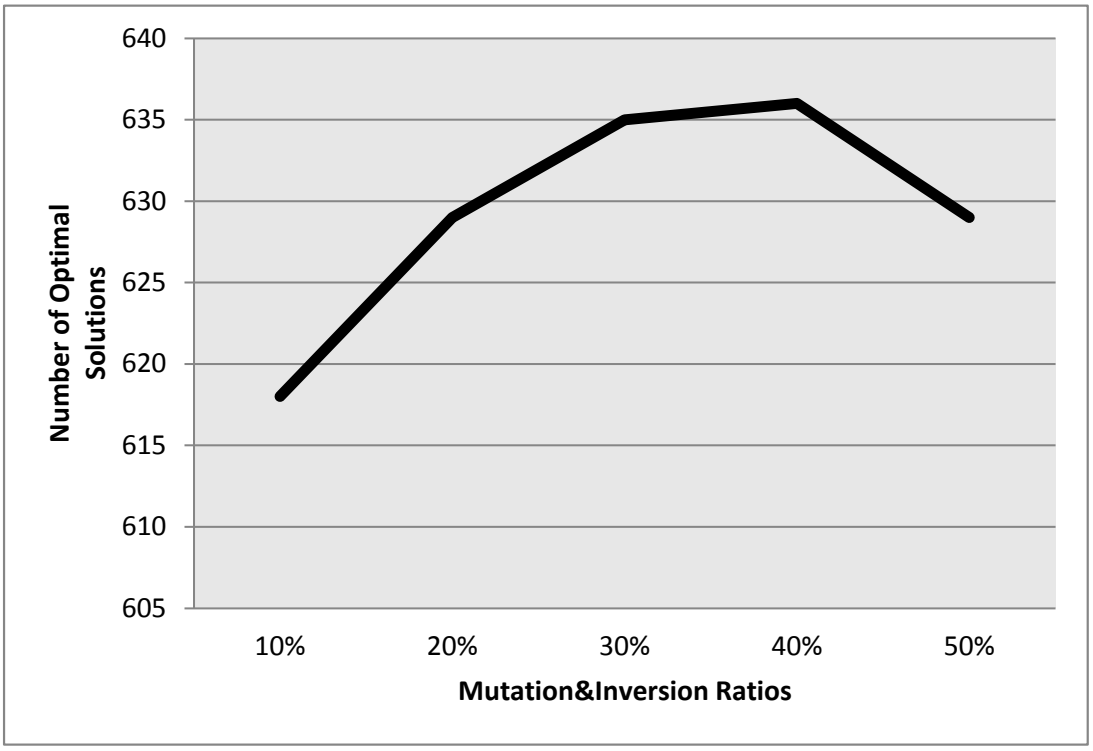

Figure 4.23: Mutation&Inversion Ratio vs. Number of Optimal Solutions Set_1(720 Problem Instances)

The results for the Set_2 data set are presented in Table 4.20 and the chart as "Crossover Ratio vs. Number of Optimal Solutions" is shown in Figure 4.24 and "Mutation&Inversion Ratio vs. Number of Optimal Solutions" is shown in 4.25 .

Table 4.20: The Effect of Changing the Crossover and Mutation&Inversion Ratio for Set_2 Data Set(480 Problem Instances)

Crossover Ratio (% of population)	# of Optimal Solutions	# of Extra Bins	Time (Sec)
20%	337	200	910.71
40%	377	123	1152.38
50%	384	118	1288.54
60%	395	97	1444.57
80%	400	90	1538.93

Mutation&Inversion Ratio	# of Optimal Solutions	# of Extra Bins	Time (Sec)
10%	398	93	1374.16
20%	400	90	1538.93
30%	399	91	2200.35
40%	398	94	2302.59
50%	403	89	2443.13

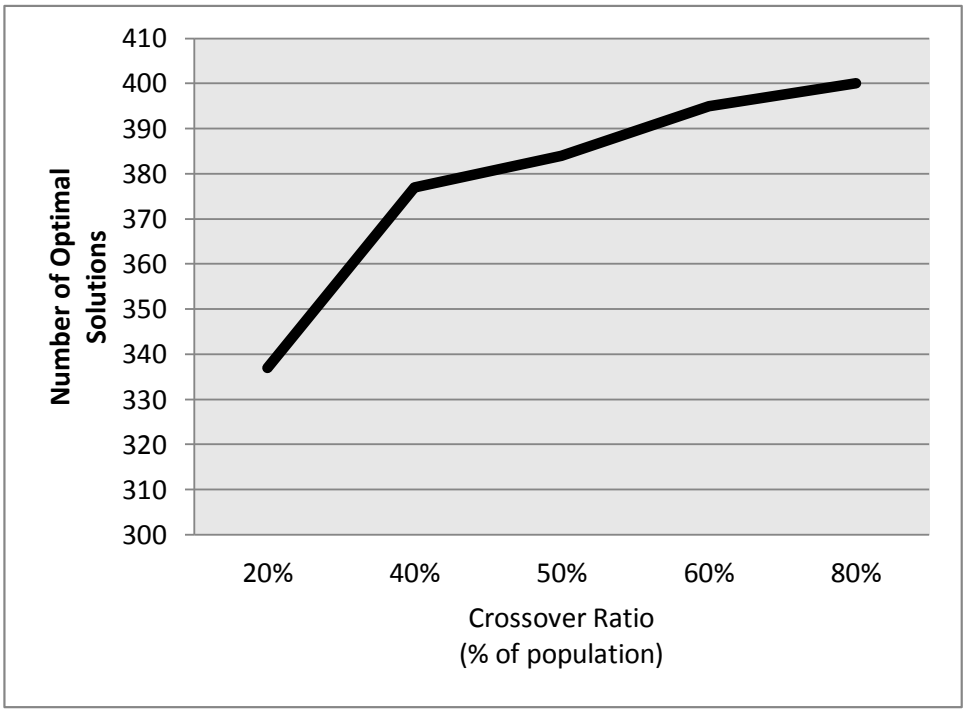

Figure 4.24: Crossover Ratio vs. Number of Optimal Solutions Set_2(480 Problem Instances)

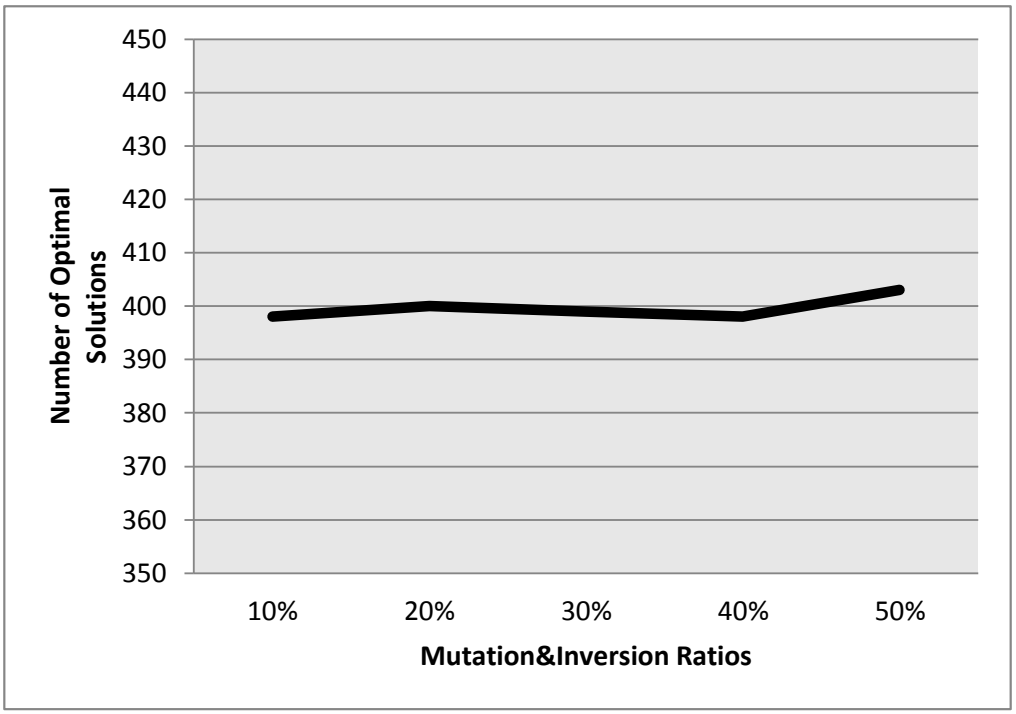

Figure 4.25: Mutation&Inversion Ratio vs. Number of Optimal Solutions Set_2(480 Problem Instances)

The results for the Set_3 data set are presented in Table 4.21 and the chart as "Crossover Ratio vs. Total Number of Extra Bins" is shown in Figure 4.26 and "Mutation&Inversion Ratio vs. Total Number of Extra Bins" is shown in 4.27 .

Table 4.21: The Effect of Changing the Crossover and Mutation&Inversion Ratio for Set_3 Data Set(10 Problem Instances)

Crossover Ratio (% of population)	# of Optimal Solutions	# of Extra Bins	Time (Sec)
20%	0	78	40.12
40%	0	74	40.64
50%	0	72	40.41
60%	0	78	49.57
80%	0	60	48.01

Mutation&Inversion Ratio	# of Optimal Solutions	# of Extra Bins	Time (Sec)
10%	0	60	41.97
20%	0	60	47.47
30%	0	77	50.29
40%	0	77	53.07
50%	0	77	55.41

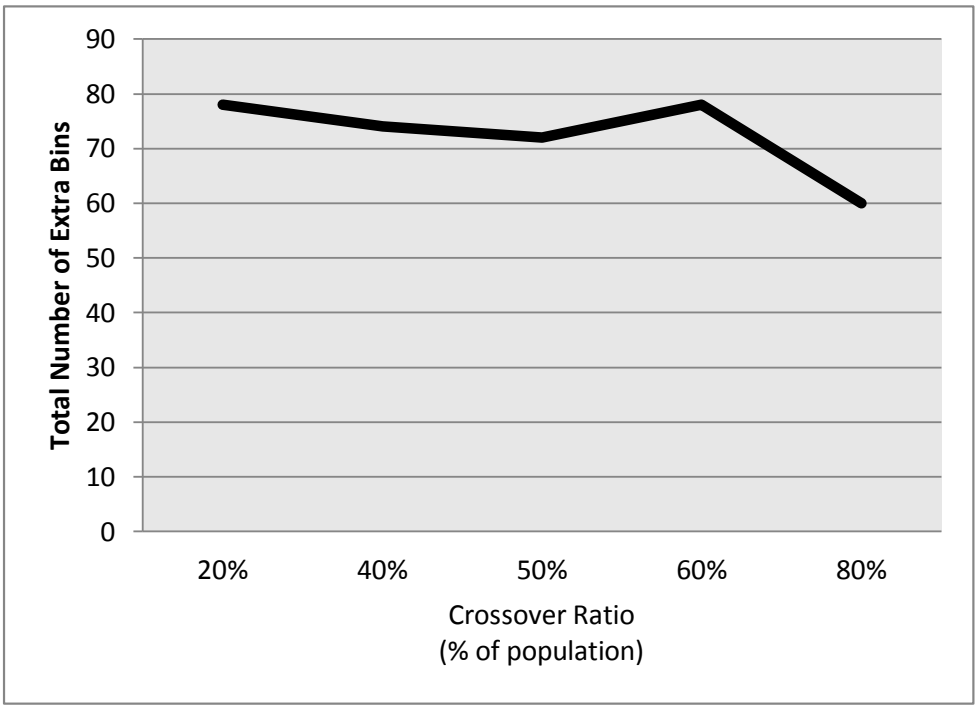

Figure 4.26: Crossover Ratio vs. Total Number of Extra Bins Set_3(10 Problem Instances)

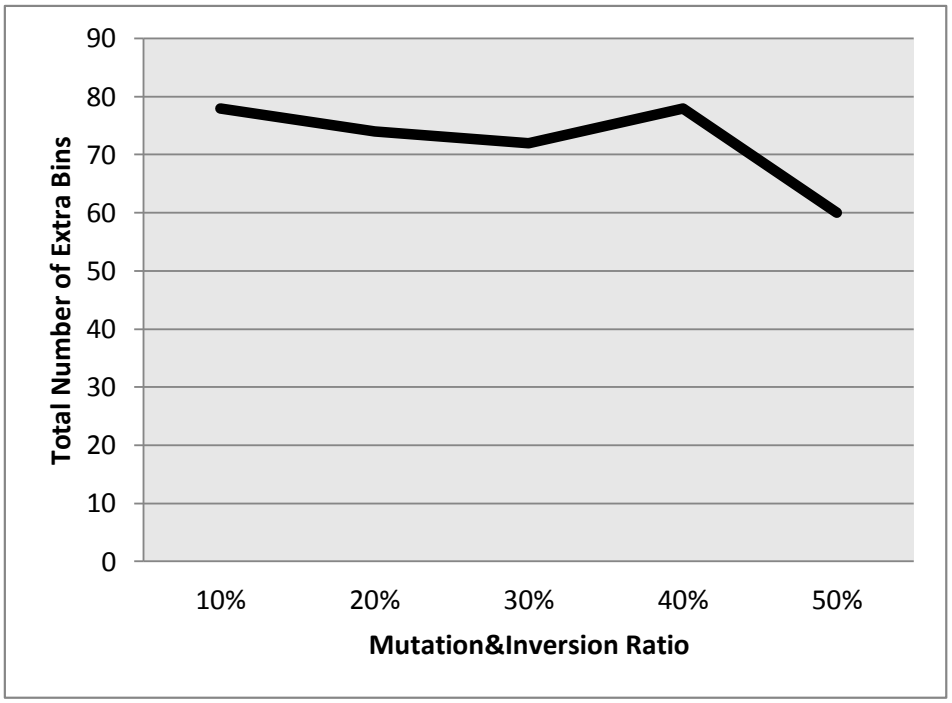

Figure 4.27: Mutation&Inversion Ratio vs. Total Number of Extra Bins Set_3(10 Problem Instances)

The results for the hard28 data set are shown in Table 4.22 and the chart as "Crossover Ratio vs. Time" is shown in Figure 4.28 and "Mutation&Inversion Ratio vs. Time" is shown in 4.29 .

Table 4.22: The Effect of Changing the Crossover and Mutation&Inversion Ratio for hard28 Data Set

Crossover Ratio (% of population)	# of Optimal Solutions	# of Extra Bins	Time (Sec)
20%	5	23	39.49
40%	5	23	44.27
50%	5	23	46.23
60%	5	23	50.74
80%	5	23	56.91

Mutation&Inversion Ratio	# of Optimal Solutions	# of Extra Bins	Time (Sec)
10%	5	23	43.26
20%	5	23	46.23
30%	5	23	46.94
40%	5	23	48.65
50%	5	23	51.10

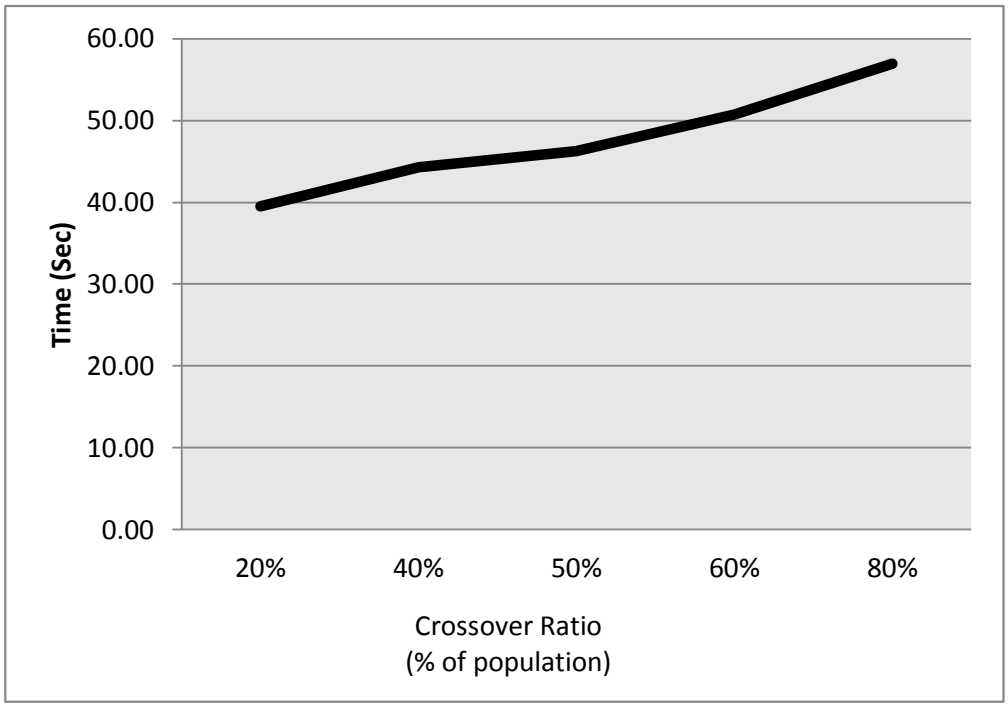

Figure 4.28: Crossover Ratio vs. Time hard28(28 Problem Instances)

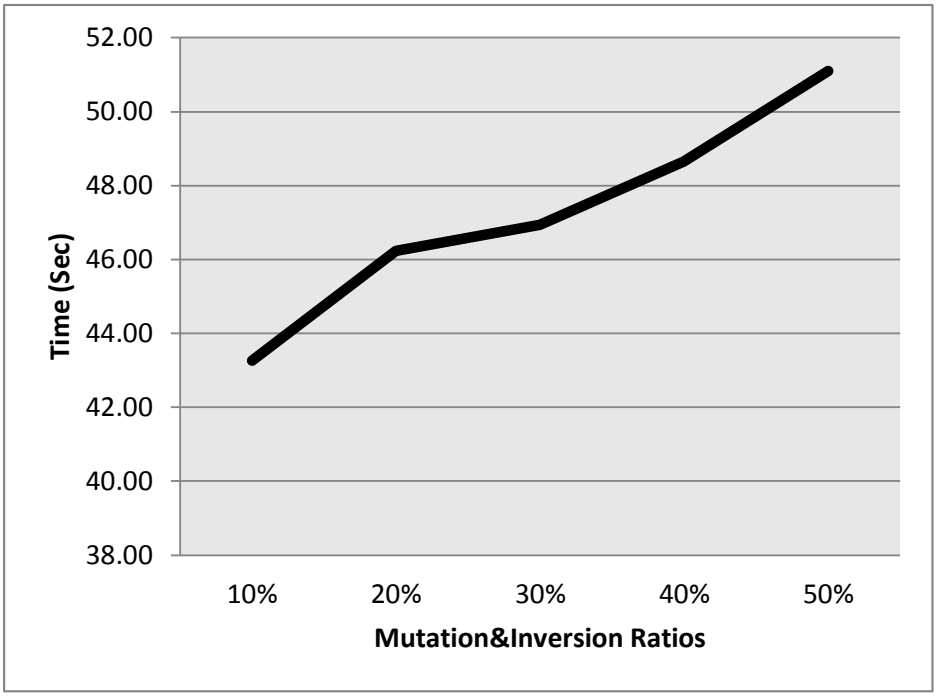

Figure 4.29: Mutation&Inversion Ratio vs. Time

The results of 4 different configurations on 4 different data sets turn out to be consistent in terms of the effects of parameters on number of optimal solutions, extra number of total bins and execution time.

4.8 Speed Up Performance of the Algorithm

In this section we compared our proposed algorithm's solutions and execution times with the sequential version of it which is used in the study by Dokeroglu and Cosar [38]. These two algorithm used the same methods as Exon Shuffling proposed by the studies [45] [12] and BFD Algorithm for reinserting remaining items in *Corssover* and *Mutatiton* and inserting chromosomes with BFD solutions to the population before the generations.

The results of the comparisons made in Set_1 are presented in Table 4.23 and the graphical visualization "Population Size vs. Time " is presented in the same Figure 4.30 for both sequential version named as CPU and GPU-parallel versions named

as GPU. Increasing *Population Size* causes increase in the execution time for both CPU and GPU implementation, but CPU implementation can't handle the increase as GPU version. The last column shows the *Speed Up Ratio* which is the ratio of Time CPU to Time GPU. There is a constant increase in the *Speed Up Ratio*. For the Set_1 data set we have not only better solution but have a speed up between 6 and 12 approximately. In addition to that increase in the *Population Size* is not have any effect on CPU implementation. We executed nearly the same algorithms in both CPU and GPU implementation. But biggest reason of this solution beat up is having a well distributed random generation of integers which help us to have a wide search space of chromosomes and its groups. In addition to have better random generator, we implemented a different switch method after crossover operations. Instead of changing the worst one with the lowest *Fitness Value* we changed *Population Size* $\times$ *Crossover Ratio* number of chromosomes in parallel. This switch method changes the order of better solutions and it yields to have a random distributed chromosomes.

Table 4.23: Comparisons between CPU and GPU Implementation for Set_1 Data Set (720 Problem Instances)

Population Size	Time CPU (Sec)	Time GPU (Sec)	# of Optimal Solutions CPU	# of Optimal Solutions GPU	Solution Ratio	Speed Up
20	4852	773	547	571	1.04	6.28
40	5907	835	547	585	1.07	7.07
60	8296	927	547	610	1.12	8.95
80	10387	999	547	612	1.12	10.40
100	12897	1014	547	613	1.12	12.72

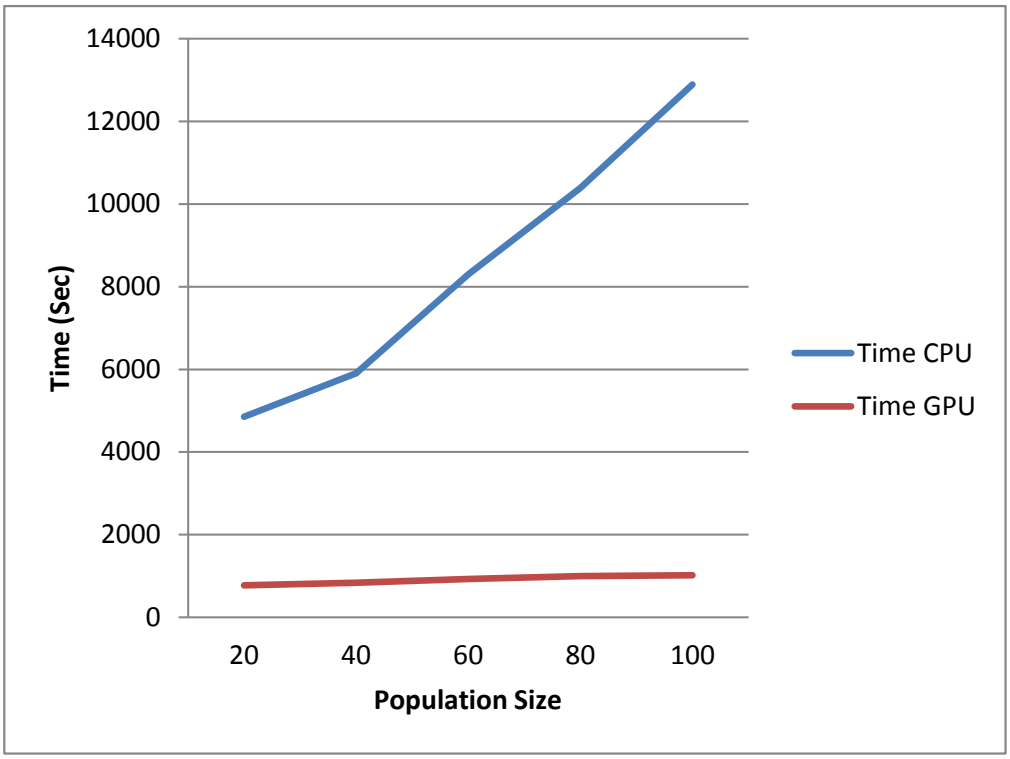

Figure 4.30: Population Size vs. Time for both CPU and GPU implementations Set_1 (720 Problem Instances)

The results of the comparisons made in Set_2 are presented in Table 4.24 and the chart shows "Number of Generations vs. Time " is presented in Figure 4.31. In this part of the study we changed the *Number of Generations* while Population Size and other paramters remain constant. As a result we noticed that Set_2 speed up fight is not harsh as Set_1 in terms of execution times. So our proposed algorithm can handle big population sizes better than big number of generations. Main cause of this fact is we parallel the operations as population size, but since generation operations need to be sequential in nature, speed up is not as effective as the test made with Set_1 data set.

Table 4.24: Comparisons between CPU and GPU Implementation for Set_2 Data Set(480 Problem Instances)

# of Generations	Time CPU (Sec)	Time GPU (Sec)	# of Optimal Solutions CPU	# of Optimal Solutions GPU	Solution Ratio	Speed Up
20	776	710	242	345	1.43	1.09
40	1610	891	240	380	1.58	1.81
60	4586	1178	243	383	1.58	3.89
80	6104	1420	244	383	1.57	4.30
100	7776	1651	242	383	1.58	4.71

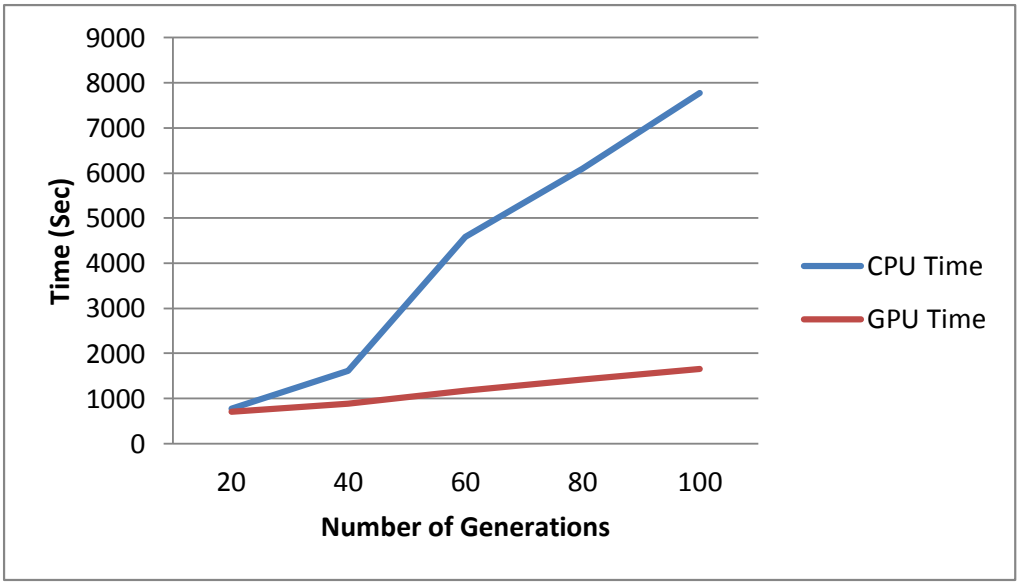

Figure 4.31: Number of Generations vs. Time for both CPU and GPU implementations Set_2(480 Problem Instances)

For Set_3 we also examined how increasing *Number of Generations* and *Population Size* effect the *Speed Up* ratio which is the ratio of Time GPU to Time CPU. In Table 4.25 we fixed the *Population Size* as "20" and tuned the *Number of Generations* between "10" and "300". *Time* and *Speed Up* ratio charted vs. *Number of Generations* in Figure4.32 and Figure4.33 . As we described above our proposed algorithm can handle big population size better. But it doesn't mean that it can't handle big number of generations as 300. Since all three generation operations executed on the GPU in parallel fashion, a jump like between 100 and 300 *Number of Generations* cause trouble in CPU implementation, it is not a big thing for our proposed algorithm. as seen in Figure4.32 .

Table 4.25: Comparisons between CPU and GPU Implementation with Changing Number of Generations for Set_3 Data Set(10 Problem Instances)

Size of Population=20

Number of Generations	Time CPU (Sec)	Time GPU (Sec)	Speed Up
10	26	4.73	5.49
20	49	6.09	8.04
40	97	10.76	9.01
60	148	15.49	9.56
80	192	15.39	12.47
100	248	25.29	9.81
300	719	72.46	9.92

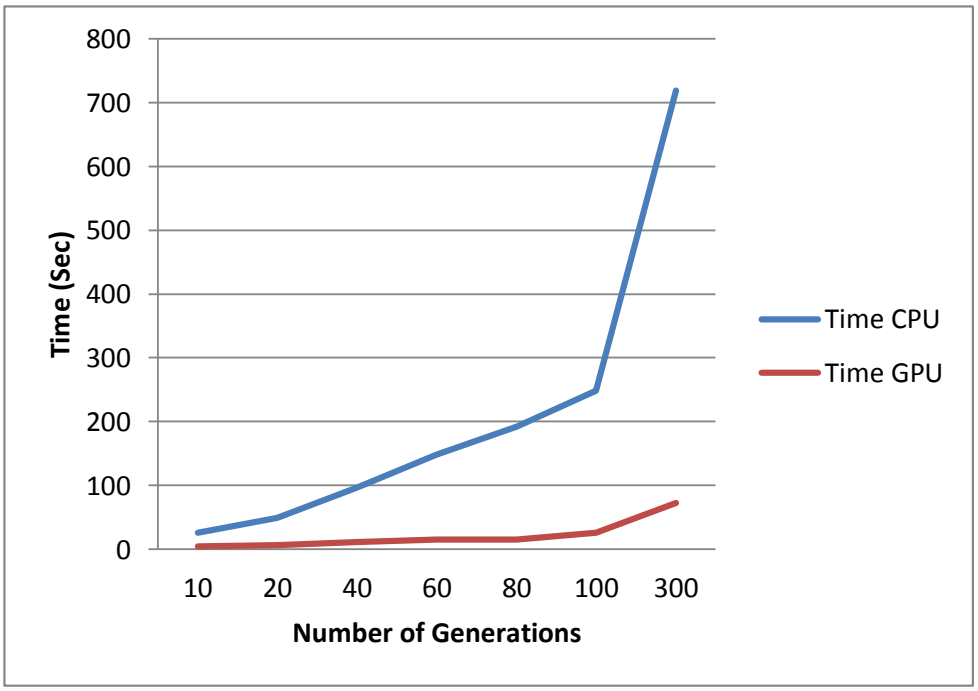

Figure 4.32: Number of Generations vs. Time for both CPU and GPU implementations Set_3(10 Problem Instances)

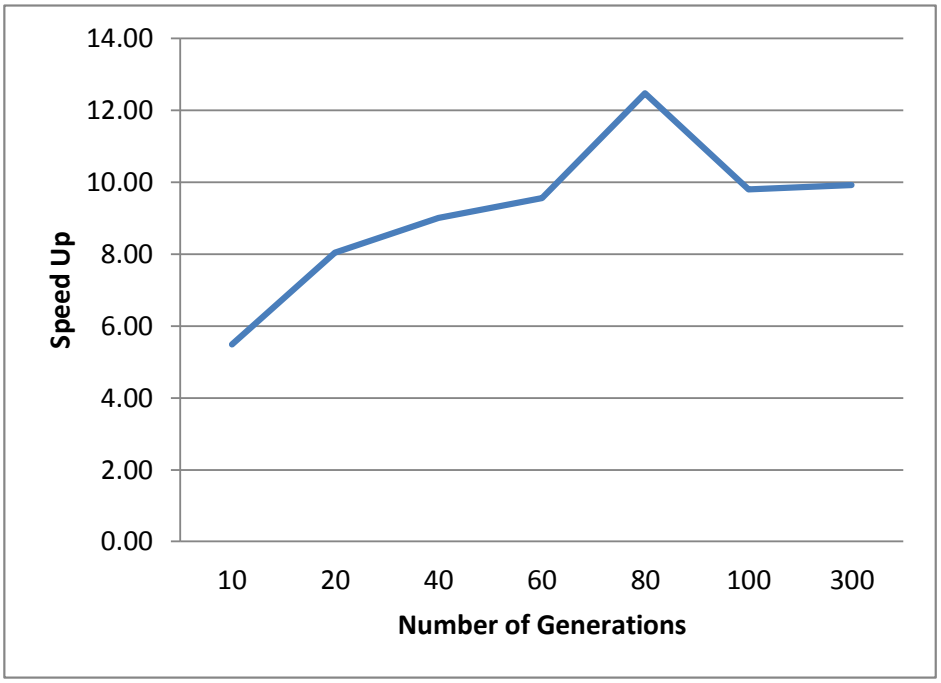

Figure 4.33: Number of Generations vs. Speed Up Ratio Set_3(10 Problem Instances)

After that, in Table 4.26 we fixed the *Number of Generations* as "40" and tuned the *Population Size* between "20" and "300". *Time* and *Speed Up* ratio charted vs. *Population Size* in Figure 4.34 and Figure 4.35 .

Table 4.26: Comparisons between CPU and GPU Implementation with Changing Population Size for Set_3 Data Set(10 Problem Instances)

Number of Generations=40

Population Size	Time CPU (Sec)	Time GPU (Sec)	Speed Up
20	148	10.92	13.56
40	193	24.38	7.92
60	290	30.67	9.46
80	394	30.85	12.77
100	486	31.40	15.48
150	726	22.75	31.91
300	1434	21.58	66.47

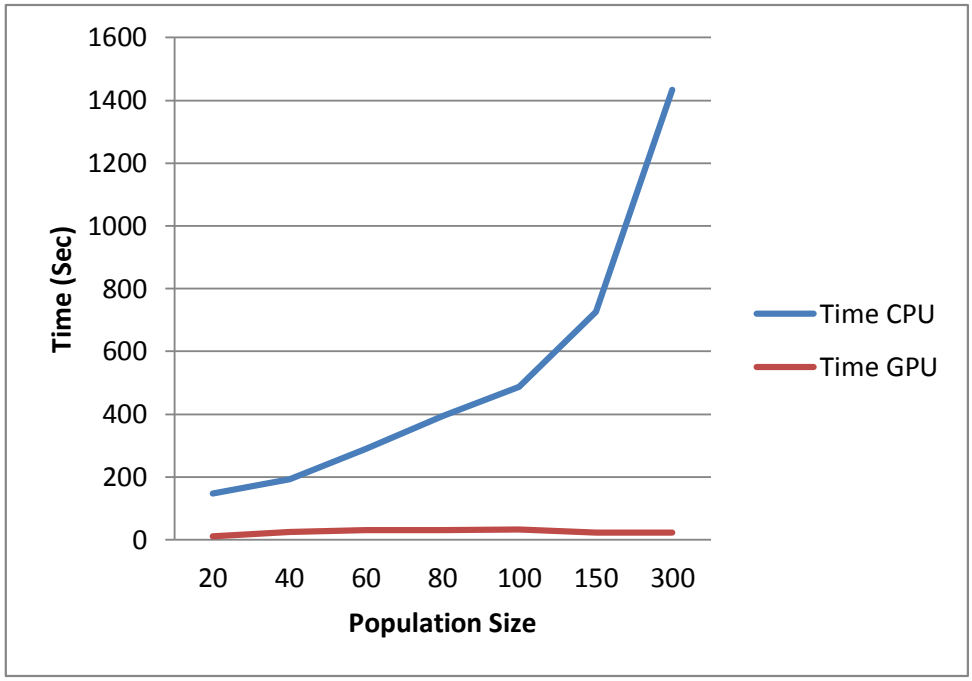

Figure 4.34: Number of Generations vs. Time for both CPU and GPU implementations Set_3(10 Problem Instances)

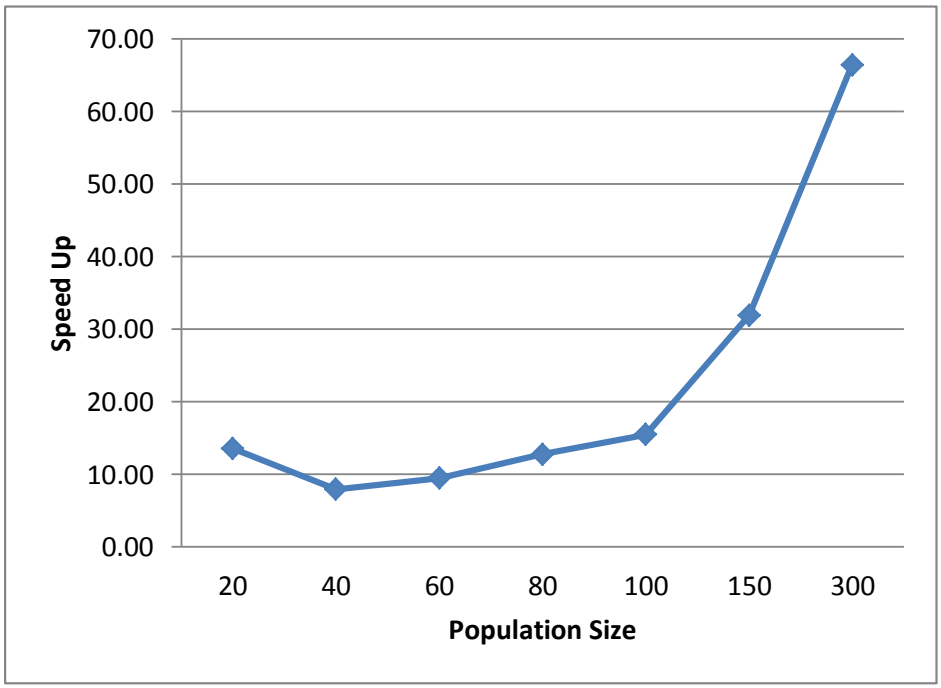

Figure 4.35: Number of Generations vs. Speed Up Ratio Set_3(10 Problem Instances)

For hard28 data set we tuned the *Population Size* and saw the change in *Speed Up* ratio in Table 4.27. We tried the show the impact in Figure 4.36 as "Population Size vs. Time" for both GPU and CPU implementation and in Figure 4.37 as "Population Size vs. Speed Up Ratio".

Table 4.27: Comparisons between CPU and GPU Implementation with Changing Population Size for hard28 Data Set(28 Problem Instances)

Population Size	Time CPU (Sec)	Time GPU (Sec)	Speed Up
20	236	38.76	6.09
40	461	48.25	9.55
60	693	40.75	17.01
80	902	41.72	21.62
100	1119	41.05	27.26
150	1693	47.05	35.99
300	3568	60.69	58.79

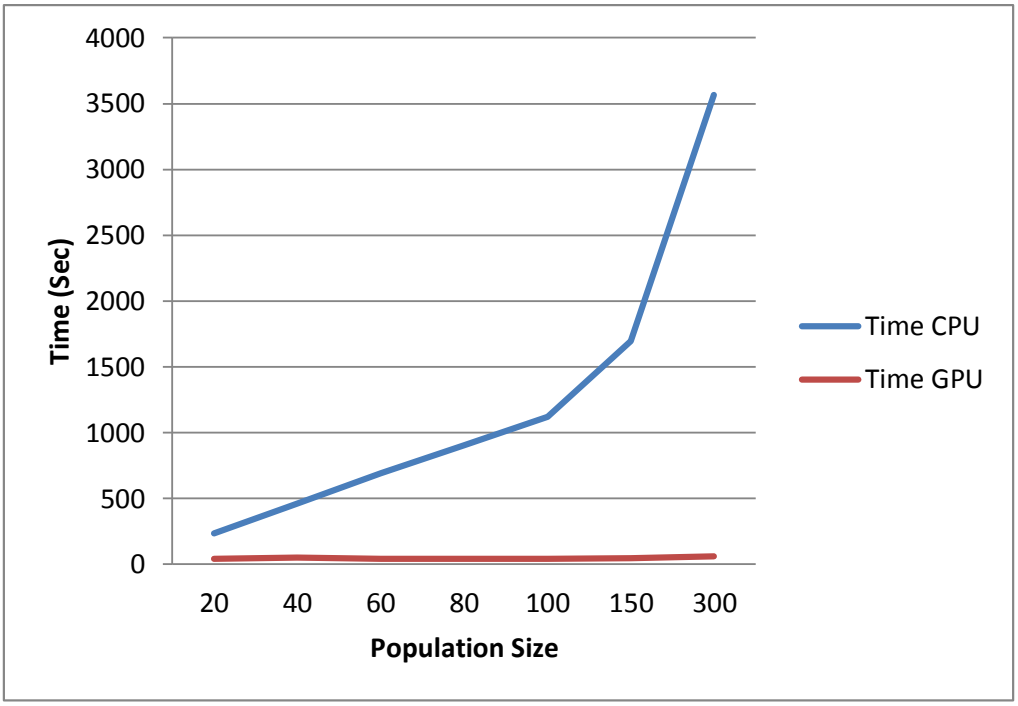

Figure 4.36: Population Size vs. Time for both CPU and GPU implementations hard28(28 Problem Instances)

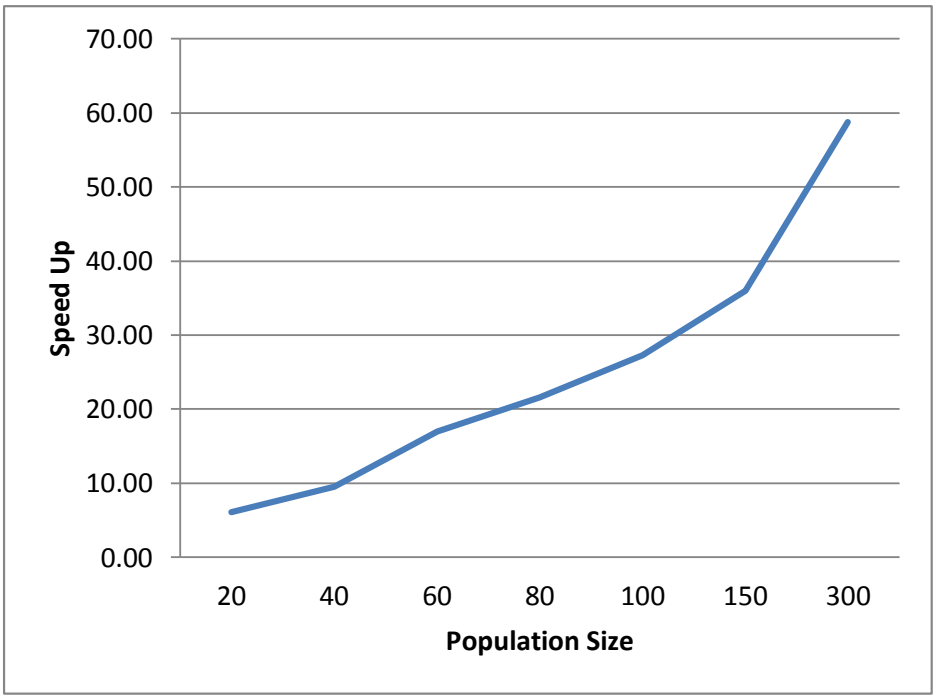

Figure 4.37: Population Size vs. Speed Up Ratio hard28(28 Problem Instances)

As seen in above Tables and Figures the biggest *Speed Up* differences occur for set_3 and hard28 data sets because of the group sizes of the chromosomes. Hard28 data set has 60-70 group sizes for the optimal solutions while Set_1 and Set_2 has 15-25 group sizes to pack all items. Since our algorithm loves to have bigger numbers bigger group sizes mean better parallelizations for specific functions like "finding slacks of the groups" and "calculating fitness of the chromosomes".

4.9 Comparison with state-of-the-art algorithms

As shown with the results and charts in this chapter, our algorithm both improves the solution quality while reducing the execution time even for a large population size and number of generations. In this section we compare our proposed algorithm with state-of-the-art algorithms in literature. Hard28 data set, one of the well known and widely preferred data set in BPP, is used for the comparisons. The execution times taken from the study which is conducted by Dokeroglu and Cosar [38] and comparison table is presented in Table 4.28 .

Table 4.28: Comparing the solution quality of GPU parallel 1DBPP-GGA-CUDA algorithm with state-of-the-art algorithms on the hard28 data set.

Algorithm	**# of Optimal Solutions**	**Time (ms.)**
BFD	2	2.26
MBS′	2	3.64
MBS	3	4.20
B2F	4	3.65
SAWMBS′	5	129.92
FFD	5	2.24
Pert-SAWMBS	5	6,946.4
Parallel Exon-MBS-BFD	5	5,341.02
1DBPP-GGA-CUDA	**5**	**7,023.56**

This comparison is not fair because we have parallel, sequential, GA and single solution versions of solutions in the same table. Yet, it may give a hint about execution times. A fair comparison can be made between Parallel Exon-MBS-BFD algorithm and our proposed 1DBPP-GGA-CUDA algorithm. The most attractive outcome of this table is we can have nearly the same execution time with an algorithm running

on expensive cluster computer(500K+ USD), only with a GPU with a cost of $200.

CHAPTER 5

CONCLUSION AND FUTURE WORK

The paradigm of heterogeneous CPU computation combined with a graphics processing unit (GPU) makes use of parallel computation to speed up scientific, engineering, consumer, and similar computation intensive applications. Since its introduction in 2007 by NVIDIA, GPU has become a part of power energy-efficient data centers in government labs, universities, enterprises, and small-and-medium businesses around the world. GPU computing presents high performance by sending computation-intensive portions of the application to the GPU card, while the rest of the code is still running on the CPU part. A contemporary CPU architecture has a few cores optimized for sequential processing while a GPU has a massively parallel architecture consisting of hundreds of smaller, more efficient cores designed for running many tasks simultaneously.

One-dimensional Bin Packing Problem (BPP) is an NP-Hard combinatorial optimization problem and created when searching solutions for real life problems. When it is not possible to obtain an exact solution due to the computation limitations for BPP, Metaheuristics/heuristics are implemented for solving large problem instances of BPP in reasonable running times with an urge to have (near-) optimal solutions. In this study, we propose a scalable heterogeneous computation based hybrid parallel algorithm (CUDA GGA for 1DBPP (CUDA-GGA-1DBPP)) that take advantage of parallel computation technique, CUDA, evolutionary grouping genetic metaheuristics, and bin-oriented heuristics to obtain high quality solutions for large scale one-dimensional BPP instances. 1,238 benchmark problems are tested with our algorithm and at the end of our exprements, we found out that for 84.57% of the problem in-

stances, optimal solutions can be seen within reasonable times comparing its sequential counterpart while finding 250 additional bins in total for the problems that optimal solution couldn't be reached. With the faster computation talent of the GPU-enabled algorithm, it is now possible to reach very large number of generations and improve the computation time of the crossover and mutation operators significantly. Existing state-of-the-art heuristics are compared with our algorithm and, we found out that the developed heterogeneous CUDA-parallel GGA can be considered as one of the best one-dimensional BPP algorithms in terms of solution quality and execution time. In addition to the better solution quality we have a speed up to 66.47 times depending on the configuration and the examined data set.

The future of parallel computation has still several opportunities to offer for the computer engineers and scientists. Most of the optimization algorithms and tools of the engineers are single-processor oriented. We believe that with the new upcoming technologies, the hard computational problems of the past century will be solved optimally while our new future is introducing harder ones.

REFERENCES

[1] Alvim, A.C.F., Ribeiro, C.C., Glover, F., Aloise, D.J., (2004). A hybrid improvement heuristic for the one-dimensional bin packing problem. *Journal of Heuristics*, 10 (2), 205-229.

[2] Garey, M.R., Johnson, D.S., (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman.

[3] Johnson, D.S., Demers, A., Ullman, J.D., Garey, M. R., Graham, R.L., (1974). Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM Journal on Computing*, 3(4), 299-325.

[4] Cantu-Paz E., (2000). *Efficient and Accurate Parallel Genetic Algorithms*, Kluwer Academic Publishers.

[5] Camacho, E.L., Terashima-Marin, H., Ochoa, G., Conant-Pablos, S.E. (2013). *Understanding the structure of bin packing problems through principal component analysis*. International Journal of Production Economics, 145(2), 488-499.

[6] Cheng, J., Grossman, M., McKercher, T. (2014). Professional Cuda C Programming. *John Wiley & Sons*.

[7] Fleszar, K., Hindi, K.S., (2002). New heuristics for one-dimensional bin-packing. *Computers and Operations Research* 29 (7), 821-839.

[8] Fleszar, K., Charalambous, C., (2011). Average-weight-controlled bin-oriented heuristics for the one-dimensional bin-packing problem, *European Journal of Operational Research* 210: 176-184.

[9] Johnson, D. S. (1973). Near-optimal bin packing algorithms (Doctoral dissertation, Massachusetts Institute of Technology).

[10] Holland, J.H., (1975). Adaptation in Natural and Artifical Systems. University of Michigan Press, Ann Arbor, MI.

[11] Luque G., Alba E., (2011). *Parallel Genetic Algorithms, Theory and Applications*, Springer.

[12] Kolkman, J. A., Stemmer, W. P. C. (2001). *Directed evolution of proteins by exon shuffling*. Nature Biotechnology, 19, 423–428.

[13] Mitchell, M., (1996). *An Introduction to Genetic Algorithms*. MIT Press.

[14] Zitzler, E., Thiele, L., (1999). Multiobjective evolutionary algorithms: A comparative case study and the strength pareto approach. *Evolutionary Computation, IEEE Transactions*, 3(4), 257-271.

[15] Dokeroglu, T. (2015). Hybrid teaching–learning-based optimization algorithms for the Quadratic Assignment Problem. Computers Industrial Engineering, 85, 86-101.

[16] Ahuja, R. K., Orlin, J. B., Tiwari, A. (2000). A greedy genetic algorithm for the quadratic assignment problem. Computers Operations Research, 27(10), 917-934.

[17] Misevicius, A. (2015). An extension of hybrid genetic algorithm for the quadratic assignment problem. Information Technology and Control, 33(4).

[18] Falkenauer, E., (1994). A new representation and operators for GAs applied to grouping problems. **Evolutionary Computation**, 2(2), 123-144.

[19] Falkenauer, E., (1996). A hybrid grouping genetic algorithm for bin packing. *Journal of Heuristics* 2 (1), 5-30.

[20] Quiroz-Castellanos, M., Cruz-Reyes, L., Torres-Jimenez, J., Gómez, C., Huacuja, H. J. F., & Alvim, A. C. (2015). A grouping genetic algorithm with controlled gene transmission for the bin packing problem. *Computers & Operations Research* 55, 52-64.

[21] Sivaraj, R., and Ravichandran, T., (2011). "An Efficient Grouping Genetic Algorithm." *International Journal of Computer Applications* 21.7 : 38-42.

[22] Grosso, P. B. (1985). Computer simulations of genetic adaptation: Parallel subcomponent interaction in a multilocus model.

[23] Pettey, C. B., Leuze, M. R., & Grefenstette, J. J. (1987). Parallel genetic algorithm. In Genetic algorithms and their applications: proceedings of the second International Conference on Genetic Algorithms: July 28-31, 1987 at the Massachusetts Institute of Technology, Cambridge, MA.

[24] Sanders, J., & Kandrot, E. (2010). CUDA by example: an introduction to general-purpose GPU programming. *Addison-Wesley Professional*.

[25] website <docs.nvidia.com/cuda/> [Last Accessed on September 17th, 2015].

[26] NVIDIA GeForce GTX 750 Ti White Paper available online on <http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce-GTX-750-Ti-Whitepaper.pdf> [Last Ac-

cessed on September 23th, 2015].

[27] EVGA NVIDIA GeForce GTX 750 Ti Product Page available online on <http://www.evga.com/Products/Product.aspx?pn=02G-P4-3755-KR> [Last Accessed on September 28th, 2015]

[28] Bhatia, A.K., Basu, S.K., (2004). Packing bins using multi-chromosomal genetic representation and better fit heuristic. *Lecture Notes in Computer Science*, 3316, 181-186.

[29] Boyer, V., El Baz, D., & Elkihel, M. (2011, February). Dense Dynamic Programming on Multi GPU. *PDP*, 545-551.

[30] Božejko, W., Hejducki, Z., Uchroński, M., Wodecki, M., Solving the flexible job shop problem on multi-GPU, *Proceedings of the International Conference on Computational Science*, Vol. 9 (0), 2020–2023.

[31] Brusco, Michael J., Hans Friedrich Köhn, and Douglas Steinley (2013). Exact and approximate methods for a one-dimensional minimax bin-packing problem. *Annals of Operations Research* 206 (1) , 611-626.,

[32] Bukata, L., Ščha, P., Hanzálek, Z. (2015). Solving the Resource Constrained Project Scheduling Problem using the parallel Tabu Search designed for the CUDA platform. *Journal of Parallel and Distributed Computing*, 77, 58-68.

[33] Cekmez, U., Ozsiginan, M., & Sahingoz, O. K. (2013). Adapting the ga approach to solve traveling salesman problems on cuda architecture. *In Computational Intelligence and Informatics (CINTI), 2013 IEEE 14th International Symposium*, 423-428.

[34] Valério de Carvalho, J.M., (1999). Exact solution of bin-packing problems using column generation and branch-and-bound. *Annals of Operations Research*, 86, 629-659.

[35] Czapiński, M. (2013). An effective parallel multistart tabu search for quadratic assignment problem on CUDA platform. *Journal of Parallel and Distributed Computing*, 73(11), 1461-1468.

[36] De la Rosa, R., Castillo, H., Zavala, J.C., Martínez, A., Estrada, H., (2015). Application of prime numbers to solve complex instances of the bin packing problem. *Proceedings of the International Conference on Numerical Analysis and Applied Mathematics* 1648, 820006.

[37] Delévacq, A., Delisle, P., Gravel, M., Krajecki, M.,(2013) Parallel ant colony optimization on graphics processing units. *Journal of Parallel and Distributed Computing*, 73(1), 52-61.

[38] Dokeroglu, T., & Cosar, A. (2014). Optimization of one-dimensional Bin Packing Problem with island parallel grouping genetic algorithms. *Computers & Industrial Engineering*, 75, 176-186.

[39] Gupta, J.N.D., Ho, J.C., (1999). A new heuristic algorithm for the one-dimensional binpacking problem. *Production Planning and Control* 10 (6), 598-603.

[40] Janiak, A., Janiak, W. A., & Lichtenstein, M. (2008). Tabu Search on GPU. J. UCS, 14 (14), 2416-2426.

[41] Martello, S., Toth, P., (1990). Knapsack Problems. Wiley, available online on <http://www.or.deis.unibo.it/knapsack.html> [Last Accessed on September 17th, 2015] .

[42] Poli, R., Woodward, J., Burke, E.K., (2007). A histogram-matching approach to the evolution of bin-packing strategies. In: *IEEE Congress on Evolutionary Computation*, 3500-3507.

[43] Pospichal, P., Jaros, J., & Schwarz, J. (2010). Parallel genetic algorithm on the cuda architecture. *In Applications of Evolutionary Computation*, 442-451.

[44] Radcliffe, N., (1991). Equivalence class analysis of genetic algorithms. *Complex Systems*, 5:183-205.

[45] Rohlfshagen, P., Bullinaria, J., (2007). A genetic algorithm with exon shuffling crossover for hard bin packing problems. *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, 1365-1371.

[46] Rong, G., Liu, G., Zheng, M., Sun, A., Tian, Y., & Wang, H. (2013). Parallel gravitation field algorithm based on the CUDA platform. *J Inform Comput Sci*, 10, 3635-3644.

[47] Saito, M., & Matsumoto, M. (2013). Variants of mersenne twister suitable for graphic processors. *ACM Transactions on Mathematical Software (TOMS)*, 39(2), 12.

[48] Scholl, A., Klein, R., Jurgens, C., (1997). BISON: A fast hybrid procedure for exactly solving the one-dimensional bin packing problem. *Computers and Operations Research* 24 (7), 627-645.

[49] Schwerin, P., Wascher, G., (1997). The bin-packing problem: A problem generator and some numerical experiments with FFD packing and MTP. *International Transactions in Operational Research* 4 (5/6), 377-389.

[50] Segura, C., Segredo, E., Leon, C., (2011). Parallel Island-Based Multiobjectivised Memetic Algorithms for a 2D Packing Problem *GECCO*, July 12-16,

Dublin, Ireland.

[51] Singh, A., Gupta, A.K., (2007). Two heuristics for the one-dimensional bin-packing problem. *OR Spectrum* 29 (4), 765-781.

[52] Stawowy, A. (2008). Evolutionary based heuristic for bin packing problem. *Computers & Industrial Engineering*, 55, 465-474.

[53] Fernandez, A., Gil, C., Banos, R., Montoya, M.G., (2013). A parallel multi-objective algorithm for two-dimensional bin packing with rotations and load balancing. *Expert Systems with Applications*,40 (13): 5169-5180.

[54] Belov, G., Scheithauer, G., Mukhacheva, E.A., (2007). One-dimensional heuristics adapted for two-dimensional rectangular strip packing. *Journal of the Operational Research Society*, 59(6), 823-832.