A Novel Grouping Genetic Algorithm for the One-Dimensional Bin Packing Problem on GPU

Sukru Ozer Ozcan, Tansel Dokeroglu(✉), Ahmet Cosar, and Adnan Yazici

Computer Engineering Department of Middle East Technical University, Universities Street, 6800 Ankara, Turkey
{ozer.ozcan,tansel,cosar,yazici}@ceng.metu.edu.tr

Abstract. One-dimensional Bin Packing Problem (1D-BPP) is a challenging NP-Hard combinatorial problem which is used to pack finite number of items into minimum number of bins. Large problem instances of the 1D-BPP cannot be solved exactly due to the intractable nature of the problem. In this study, we propose an efficient Grouping Genetic Algorithm (GGA) by harnessing the power of the Graphics Processing Unit (GPU) using CUDA. The time consuming crossover and mutation processes of the GGA are executed on the GPU by increasing the evaluation times significantly. The obtained experimental results on 1,238 benchmark 1D-BPP instances show that our proposed algorithm has a high performance and is a scalable algorithm with its high speed fitness evaluation ability. Our proposed algorithm can be considered as one of the best performing algorithms with its 66 times faster computation speed that enables to explore the search space more effectively than any of its counterparts.

Keywords: 1D Bin packing · Grouping genetic · CUDA · GPU

1 Introduction

One-dimensional Bin Packing Problem (1D-BPP) is a challenging NP-Hard combinatorial problem which is used to pack finite number of items into minimum number of bins [1]. The general purpose of the 1D-BPP is to pack items of interest subject to various constraints such that the overall number of bins is minimized. More formally, 1D-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.

Although problems with a small number of items up to 30 can be solved with brute-force algorithms, large problem instances of the 1D-BPP cannot be solved exactly. Therefore, metaheuristic approaches such as genetic algorithms (GA), particle swarm, tabu search, and minimum bin slack (MBS) have been

© The Author(s) 2016
T. Czachórski et al. (Eds.): ISCIS 2016, CCIS 659, pp. 52–60, 2016.
DOI: 10.1007/978-3-319-47217-1_6

widely used to solve this important problem (near-) optimally [2–5]. Most of the state-of-the-art algorithms that have been proposed to solve the 1D-BPP are designed to run on a single processor and do not make use of the high performance computation opportunities that are offered by the recent parallel computation technologies. In this study, introduce an efficient Grouping Genetic Algorithm (GGA) by making use of the Graphics Processing Unit (GPU) using Compute Unified Device Architecture (CUDA) [6–9]. The population of solutions is kept on memory of GPU and the time consuming crossover, mutation, and fitness evaluation processes of the proposed GGA are also executed on the GPU. Therefore, a high performance heterogeneous computing environment is provided with a parallel computation support of GPU [10,11]. Our proposed algorithm is tested on 1,238 benchmark problem instances and has been observed to be a robust and scalable algorithm that can be considered as one of the best performing algorithms with its up to 66 times faster computation speed than the CPU-based version of GGAs. This talent of our proposed algorithm enables it to explore the solution space more effectively than any of its single-processor versions and obtain (near-)optimal results.

2 Proposed Algorithm (1D-BPP-CUDA)

Falkenaur's chromosome structure is chosen for our study due to its high performance [6,7].

Exon Shuffling Crossover: We use exon shuffling crossover [12], a recent technique borrowed from molecular genetics, for our proposed parallel algorithm. Molecular genetics is the field of biology and genetics that studies genes at a molecular level and employs methods to elucidate the molecular function and interactions among genes. An offspring is generated by a two phase crossover. In the first phase, all mutually exclusive segments are combined. In the second phase, the remaining items are used to build a new bin. During the execution of the algorithm, the exon shuffling crossover operations are run on the GPU.

The Mutation operator: enables new solutions using the current optimal solution. In this study, the mutation operator works based on the predefined mutation ratio. The number of groups chosen change depending on the population size and mutation ratio. The mutation operator works on a number of groups computed as multiplication of population size and the mutation ratio and select a 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.

Inversion operator: is applied to increase the transfer probability of fitter gene pair to the next generation. At the beginning of process, selected groups are interchanged [6]. 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.

Fitness function: gives us a value that is based on an equation defined by Falkenauer given below:

$$FF = \sum_{i=1}^{nb} \left(\frac{F_i}{c} \right)^k \tag{1}$$

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. nb is the number of bins, F_i is the sum of weights of the elements packed into the bin i (i = 1 ,..., nb), c is the bin capacity, and k is a heuristic exponential factor. The value k expresses a concentration on the almost full bins in comparison to less filled ones. Falkenauer used $k = 2$ but Stawowy reported that $k = 4$ gives slightly better results therefore, we prefer the second value [15].

For calculating the fitness values of each chromosome, we prefer to have an enough block size division of size of population by 64 and 64 threads. So every chromosome's fitness value is calculated by concurrent blocks and threads. 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.

For the mutation and generation of initial population, we need to generate integer random numbers. We use CURAND library of GPU side for this process. A basic generation of CURAND is used in our study. We send the state pointers to kernels to make the states ready for the generation-kernels. In this study, we use 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. [13]. The other state we used is CURAND's default state which generates an array of pseudorandom numbers greater than 2^{190}. 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 are used for the generation, and the last one is used for asynchronous memory copy of item weights from host to device. These three operations are completely independent and run asynchronously.

An initial population is generated with the random numbered arrays for the proposed algorithm. After allocation enough memory on the device the kernel which executes the generation procedure, is launched. After the generation of each chromosome, the population array is filled with the chromosomes resident on the device memory.

Generating an initial population that is larger than the population that you will be working on by executing generations and pruning its size by selecting the fittest individuals is a very effective way for GA. With this method, it is possible to start with a higher quality population. This is called truncation. In our proposed algorithm, we applied this method on GPU. A number of random individuals are generated on the GPU and sent to CPU memory. CPU side code selected the best individuals by pruning the all initial population with a truncation ratio. The high-quality population is sent back to the GPU to be improved through the generations.

BFD is one of the simplest and high performance algorithms for solving the 1D-BPP. In our proposed algorithm, crossover and mutation operators use a BFD heuristic to reinsert the remaining items [4].

3 Performance Evaluation of Experimental Results

The PC used during the experiments has Intel Core i5-2467M CPU 1.60 GHz with 4 cores, 4 GB Memory (RAM), 64 bit Windows 7 Operating System, and EVGA NVIDIA GeForce GTX 750 Ti GPU (a mid-sized GPU designed for both gaming and computing environment).

Four different sets of problem instances are used during the experiments. The problem instances are set_1, set_2, set_3 [14] and hard28 [16] (Table 1).

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 of each can be different for each configuration therefore, we set the best block and thread sizes to have a reasonable execution time.

The results of (near-)optimal population size for the Set_1 data set are presented in Table 2 (Bold face numbers are selected as the optimal solution, 80 individuals). *# 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. It is observed

Table 1. Information about the problem instances

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}

Table 2. The effect of changing population size for Set_1 data set (# of generations is 40, truncate ratio is 20, mutation ratio is 0.2, inversion ratio is 0.2)

population size	# of optimal solutions	# of extra bins	execution 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	611	3712.35

that increase in population size has a limited effect on number of optimal solutions when number of generations is constant. The optimal number of population is selected for the remaining problem sets as it is performed on Set_1.

After finding the best population size for the algorithm, we performed tests on the number of generations to observe how it effects the solution quality and execution time of the algorithm. 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 the 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 3.

Mutation and inversion ratios correspond to the size of the array that will be generated in mutation and inversion processes. We tried to select the most effective ratios to find (near-)optimal solutions. The number of optimal solutions has an increasing pattern for Set_1 and Set_2 data sets. Additionally, an optimal number of solution 5 and extra number of bins 23 are found as a result for hard28 data set.

Table 3. The effect of changing the number of generations for Set_1 data set (# of population is 80, truncate ratio is 20, crossover ratio is 0.5, mutation ratio is 0.2, and inversion ratio is 0.2)

# of generations	# of optimal solutions	# of extra bins	execution 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

Table 4. Comparisons between CPU and GPU implementation for Set_1 data set

population size	CPU-based exec.time	GPU-based exec.time	CPU solutions	GPU solutions	speed-up ratio
20	4852	773	547	571	6.28
40	5907	835	547	585	7.07
60	8296	927	547	610	8.95
80	10387	999	547	612	10.40
100	12897	1014	547	613	12.72

Table 5. Comparisons between CPU and GPU implementation for hard28 data set

population size	CPU-based exec.time	GPU-based exec.time	speed-up ratio
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**

The results of the comparisons made on problem Set_1 are presented in Table 4 for both CPU and GPU-parallel versions. Increasing the Population Size causes increase in the execution time for both CPU and GPU versions. The last column of Table 4 shows the Speed-Up Ratio. There is a constant increase in the *Speed Up Ratio*. For the data set_1, we have not only better solutions but have a speed up nearly 12 times approximately. In addition to that increase in the Population Size it does not have any effect on CPU implementation. The most important reason of this is to have a well distributed random generation of integers which provides us a wider search space of chromosomes and its groups.

Table 5 presents the speed-up performance of the proposed algorithm for the hard28 problem instances. The speed-up ratio is observed to be 66.47 for the problem set. The 1D-BPP-CUDA algorithm terminates the execution of the generations when it finds the optimal solution of the problem instance otherwise, it continues to search the solution space through larger number of generations. Therefore, the speed-up value of the algorithm is observed to be the highest on the problem set hard28 where obtained number of optimal solutions is less than the other problem sets and the number of generations are performed much more than the other problem sets.

As shown in the results, 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

Table 6. Comparing the solution quality of GPU parallel 1D-BPP-GGA-CUDA algorithm with state-of-the-art algorithms on the hard28 data set

Algorithm	# of optimal solutions	Time (ms.)
BFD	2	2.3
MBS$'$	2	3.6
MBS	3	4.2
B2F	4	3.6
FFD	5	2.2
SAWMBS$'$	5	129.9
Pert-SAWMBS	5	6,946.4
Parallel Exon-MBS-BFD	5	5,341.0
1D-BPP-CUDA	**5**	**7,023.6**

preferred data set in BPP, is used for the comparisons [4]. See Table 6 for the results. This comparison may seem unfair however, 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 1D-BPP-CUDA algorithm.

With the (near-)optimal parameter settings of the 1D-BPP-GGA-CUDA algorithm, 84.57 % of the problem instances are solved optimally and the solutions found for each of the remaining problem instances produced only a single extra bin, which can be considered as high performance when compared with the sate-of-the-art algorithms.

4 Conclusions and Future Work

In this study, we propose a scalable heterogeneous computation based algorithm (1D-BPP-CUDA) that take advantage of CUDA, evolutionary grouping genetic metaheuristics, and bin-oriented heuristics to obtain high quality solutions for large scale 1D-BPP instances. A total number of 1,238 benchmark problems are examined with the proposed algorithm and it is shown that optimal solutions for 84.57 % of the problem instances can be obtained within practical optimization times while solving the rest of the problems with no more than one extra bin (250 additional bins in total). In addition to the higher solution quality, we have a speed-up of 66.47 times depending on the examined data set. When the results are compared with the existing state-of-the-art heuristics, the developed parallel hybrid grouping genetic algorithms can be considered among the best 1D-BPP algorithms in terms of computation time and solution quality.

References

1. Fleszar, K., Hindi, K.S.: New heuristics for one-dimensional bin-packing. Comput. Oper. Res. **29**(7), 821–839 (2002)
2. Cantu-Paz, E.: Efficient and Accurate Parallel Genetic Algorithms. Kluwer Academic Publishers, Dordrecht (2000)
3. Holland, J.H.: Adaptation in Natural and Artifical Systems. University of Michigan Press, Ann Arbor (1975)
4. Dokeroglu, T., Cosar, A.: Optimization of one-dimensional Bin Packing Problem with island parallel grouping genetic algorithms. Comput. Ind. Eng. **75**, 176–186 (2014)
5. Fernandez, A., Gil, C., Banos, R., Montoya, M.G.: A parallel multi-objective algorithm for two-dimensional bin packing with rotations and load balancing. Expert Syst. Appl. **40**(13), 5169–5180 (2013)
6. Falkenauer, E.: A new representation and operators for GAs applied to grouping problems. Evol. Comput. **2**(2), 123–144 (1994)
7. Falkenauer, E.: A hybrid grouping genetic algorithm for bin packing. J. Heurist. **2**(1), 5–30 (1996)
8. Quiroz-Castellanos, M., Cruz-Reyes, L., Torres-Jimenez, J., Gmez, C., Huacuja, H.J.F., Alvim, A.C.: A grouping genetic algorithm with controlled gene transmission for the bin packing problem. Comput. Oper. Res. **55**, 52–64 (2015)
9. Sivaraj, R., Ravichandran, T.: An efficient grouping genetic algorithm. Int. J. Comput. Appl. **21**(7), 38–42 (2011)
10. Zitzler, E., Thiele, L.: Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach. IEEE Trans. Evol. Comput. **3**(4), 257–271 (1999)
11. Dokeroglu, T.: Hybrid teaching-learning-based optimization algorithms for the Quadratic Assignment Problem. Comput. Ind. Eng. **85**, 86–101 (2015)
12. Rohlfshagen, P., Bullinaria, J.: A genetic algorithm with exon shuffling crossover for hard bin packing problems. In: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation, pp. 1365–1371 (2007)
13. Saito, M., Matsumoto, M.: Variants of mersenne twister suitable for graphic processors. ACM Trans. Math. Softw. (TOMS) **39**(2), 12 (2013)
14. Scholl, A., Klein, R., Jurgens, C.: BISON: A fast hybrid procedure for exactly solving the one-dimensional bin packing problem. Comput. Oper. Res. **24**(7), 627–645 (1997)

15. Stawowy, A.: Evolutionary based heuristic for bin packing problem. Comput. Ind. Eng. **55**, 465–474 (2008)
16. Belov, G., Scheithauer, G., Mukhacheva, E.A.: One-dimensional heuristics adapted for two-dimensional rectangular strip packing. J. Oper. Res. Soc. **59**(6), 823–832 (2007)