Applied Mathematics & Information Sciences
An International Journal

http://dx.doi.org/10.18576/amis/130411

Enhanced Data Parallelism for Irregular Memory Access Optimization on GPU

Tamizharasan P. S.[*] and Ramasubramanian N.

Department of CSE, National Institute of Technology, Tiruchirappalli, India

Received: 15 Oct. 2018, Revised: 29 Jan. 2019, Accepted: 17 Mar. 2019
Published online: 1 Jul. 2019

Abstract: The use of many-core architectures improves the performance of most of the data-intensive applications. One of the challenging tasks for modern many-core architectures is to handle the irregular memory access effectively. Unlike regular memory access applications, an increase in the size of the problem in an irregular memory access application leads to a reduction in overall performance. A mapping between on-chip and off-chip memory through the heterogeneous communication channel also poses significant challenges. In this paper, a k-exchange algorithm with ant colony optimization is proposed to improve the performance of irregular memory access applications such as Multi-dimensional Knapsack Problem (MKP) and Traveling Salesman Problem (TSP) on the Graphics Processing Units (GPU). A different set of instances of OR library and TSPLIB are considered for experiments. The obtained results show an improvement in terms of optimal solution and speedup for the MKP and TSP instances.

Keywords: MKP, TSP, GPU, irregular memory access

1 Introduction

Data-intensive applications are mapped to multi- and many-core architectures, because of the scope for massive parallelism. Graphics Processing Units (GPUs) and Many Integrated Core (MIC) are widely used to implement parallel applications using programming languages such as Compute Unified Device Architecture (CUDA), OpenMP and OpenCL [1–3]. These architectures face different challenges like mapping workload to different cores effectively, numerical stability, non-uniform access which may result in load imbalance, reuse of data and optimizing data locality [4].

With the adequate processing elements, many-core architectures perform better for the applications which involve regular memory access pattern. Mapping and optimizing irregular memory access applications on GPUs require more effort than working on regular memory access applications. Irregularity in memory access and different capacity of heterogeneous communication channels demand more attention especially deep learning architectures such as Convolutional Neural Networks (CNN) and Recurrent Neural Networks (RNN). The sparse connection between the layers of the networks exhibits irregular memory access [5, 6]. Mapping and optimizing these deep learning models on many-core architectures are difficult since many challenges such as schedulability and dependency checking are involved. It is difficult to analyze the nature of memory access of all applications during design time itself. Therefore the challenges of enhancing performance and handling on-chip memory and off-chip memory bandwidth bottleneck are still open [7–9]. Many irregular applications have been mapped to GPUs for analysis and performance enhancement. Metrics like control flow irregularity and memory access irregularity are also considered for characterizing irregular applications for many-core architectures [10]. However, effective utilization of on-chip and off-chip memory bandwidth remains a challenge for the data-intensive irregular deep learning applications. Most of the deep learning-related studies have focused on the application level to report the improvements in accuracy by trading-off the architecture level performance. In this paper, ant colony optimization with k-exchange algorithm is proposed to address multichoice multidimensional knapsack problem and traveling salesman problem considering an irregular memory access application. The paper is organized as follows. Section 2 gives the related work of multidimensional knapsack problem, traveling salesman problem and its variants. Section 3 proposes an ant colony

[*] Corresponding author e-mail: tamizh5500@mail.com

optimization with k-exchange algorithm. Section 4 provides experimental results and finally, section 5 gives the conclusion with future aspects of the work.

2 Motivation and Related Work

Multidimensional Knapsack (MKP): For a given set of n items with weight w_i and value v_i, the maximum weight capacity being W,

maximize $\sum_{i=0}^{n} v_i x_i$

subject to $\sum_{i=0}^{n} w_i x_i \leq W$, where $x_i \in \{0,1\}$

where x_i represents the number of items in the knapsack. The objective is to yield maximum profit with the minimal use of x_i. It is a NP-hard problem used to optimize and yield better results. It has been implemented with particle swarm optimization, association rule mining and space management problem. Many solutions have been proposed for 0/1 MKP because of its use in real-time applications like financial derivatives and network planning. Knapsack problem finds the sum of different sized objects collected in such a way that the solution is less than the capacity of knapsack. This condition is followed in the basic knapsack problem whereas MKP follows more than one condition. Solving a Knapsack problem can be considered as allocating the problem into an irregular mesh graph [11]. 0/1 MKP is a variant of knapsack problem in which 0 represents an item that's not selected and 1 for selected item. Knapsack and its variants have already been mapped to multicore architectures. Extensive studies on MKP optimization problem have been carried out and it has been resolved with different approaches like metaheuristic algorithms and genetic algorithm. Combinations of different algorithms are used for further improvement in the optimization problem [12].

Variations of MKP algorithm have been dealt with limited benchmark instances which are almost of similar type. It is essential that these algorithms have to be implemented with different types of instances by considering profits and weights as key parameters. Instances can be generated by correlating profits with weights in different dimensions [13]. A scheduling algorithm has been proposed to solve knapsack problem on hypercube while considering its vertices as processors and the memory access as irregular access [11]. Different heuristic methods have also been adopted for solving knapsack problems and its variants [14]. Travelling Salesman Problem (TSP) with Max-Min Ant System (MMAS) has been used to report the performance benefit of GPU using Cg(C for graphics) in the last decade. CUDA implementations of MMAS are also used for solving other problems like satisfiability and parallel search algorithm. Various studies have reported a speedup range from 2x to 32x by using the hybrid ant system along with other algorithms [15–18].

For TSP, ant systems with roulette wheel selection-based methods are used in the tour construction phase to reduce execution time. TSP has been considered as an important NP-hard problem and the implementation of hybrid TSP and ant colony optimization solves many real-time problems [19–21]. There are studies that report improvised speedup by allocating the group of instances to thread blocks in edge-detection algorithm for image processing [22].

Many hybrid algorithms are used for real-time computational problems that incorporate Ant Colony Optimization(ACO) to provide an optimized solution [23]. Different methods such as graph-based model and multi-objective model are also used with ACO for further improvement of results. Automatic design method for ACO has also been proposed and compared for some specified conditions [24, 25]. ACO has been implemented on a variety of architectures for different applications. Parallel ant programming for classification problems gives a remarkable performance when mapped to GPUs for larger data set like UCI [26, 27].

3 MKP Optimization using k-exchange algorithm

In this paper, modifications have been proposed to optimize multidimensional knapsack problem considering it as resource allocation problem particularly for effective memory bandwidth utilization. In the transformation step, four local optimization cases with neighborhoods that involve changing up to k components of the solution are introduced. The k components are referred to as k-opt.

Algorithm 1 MKP Optimization Algorithm

```
1:  Read the solution and remaining capacity of all dimensions
2:  Create two subsets of objects $Obj_{unsel}$ and $Obj_{sel}$
3:  Select an object from the two subsets
4:  while No duplicate pair do
5:      if $Obj_{unsel} \geq Obj_{sel}$ and $constraints_{max} \in \mathbb{Z}^d$ then
6:          Identify pair of objects
7:      end if
8:  end while
9:  Discard all pairs except the best solution
10: Update the results
```

The local optimization of this procedure finds the best pattern that can be considered for effective bandwidth utilization of memory hierarchy of modern architectures. The proposed MKP optimization algorithm is given in Algorithm 1. The algorithm takes initial knapsack solution and the remaining capacity of all dimensions obtained by a heuristic-based ACO and proceeds as follows. It separates the solution from the list of objects into two subsets, one for selected objects and other for unselected objects. It identifies the pair of objects one from selected and one from unselected that should satisfy all of the maximum constraints in all dimensions along with remaining capacity from the previous solution in order for the exchange to take place. The value of an unselected object should be greater than or equal to the value of the selected object. The duplicates can be avoided, by checking the repetition of objects in more than one identified pair. If an object is present in more than one pair, all pairs are discarded except one which offers the best solution. The existing solution and the remaining capacity with the new results are updated. The four local search algorithms 1-opt, 1.5-opt1, 1.5-opt2 and 2-opt based on k-exchange are proposed to improve the solution. The configuration of four algorithms is given in Table 1. These algorithms are based on local optimization with neighborhoods that involve changing up to k components of the solution which is often referred to as k-opt.

Table 1: Different Local Search Methods

S.No.	Local Search Methods	No. of object(s) from selected list	No. of object(s) from unselected list
1	1-opt	1	1
2	1.5-opt1	2	1
3	1.5-opt2	1	2
4	2-opt	2	2

3.1 CUDA Implementation

The followings are four auxiliary kernels which are common for all proposed algorithms:

$$init\_var <<< 1,1 >>> (d\_sol\_len, d\_len\_sel, d\_len\_unsel);$$

launching 1 block with 1 thread, which initializes the device memory variables.

$$sort\_list <<< 1, h_n >>> (d\_best\_ant\_so\_far, d\_obj\_list1, d\_obj\_list2, d\_len\_sel, d\_len\_unsel, h\_n);$$

launching 1 block with n threads (equal to n objects), which separates the solution into two sub sets as mentioned in section 3.2 point 2.

$$sort\_result <<< 1, h\_sol\_len >>> (d\_sol, d\_sol\_new, d\_sol\_len);$$

launching 1 block with threads equal to the size of the pairs generated, which internally sorts the pairs based on its value using parallel bitonic sort.

$$Bitonic\_Sort\_Kernel <<< 1, N >>> (d\_sol\_new, j, k);$$

launching 1 block with threads equal to the size of the list to be sorted in parallel.

3.1.1 1-Opt

The following four kernels are specific for 1-opt local search.

$$create\_sol\_one\_opt <<< l1, l2 >>> (d\_sol, d\_sol\_len, d\_obj\_list1, d\_obj\_list2, d\_pp, d\_rr, d\_best\_ant\_so\_far, h\_m);$$

launching blocks equal to the size of selected object list with threads equal to the size of unselected object list. It identifies the object pairs as mentioned in section 3.2 point 3.

$$avoid\_duplicates\_one\_opt <<< h\_sol\_len - 1, h\_sol\_len - 1 >>> (d\_sol);$$

launching blocks equal to the size of solution generated by the above kernel call with equal number of threads. It avoids the duplicates as mentioned in section 3.2 point 4.

$$update\_best\_ant\_one\_opt\_first <<< 1, h\_m >>> (d\_best\_ant\_so\_far, d\_sol\_new, d\_rr, h\_m);$$

$$update\_best\_ant\_one\_opt\_rest <<< 1, h\_m, sizeof(unsignedint) * h\_m >>> (d\_best\_ant\_so\_far, d\_sol\_new[i], d\_rr, h\_m);$$

launching 1 block with threads equal to the number of constraints. It updates the actual solution based on the new results obtained by the local search.

3.1.2 1.5-Opt1

The following four kernels are specific for 1.5-opt1 local search:

$$dim3block\_dim(l1-1, l1-1, 1);$$

$$create\_sol\_one\_five\_opt1 <<< block\_dim, l2 >>> (d\_sol, d\_sol\_len, d\_obj\_list1, d\_obj\_list2, d\_pp, d\_rr, d\_best\_ant\_so\_far, h\_m);$$

launching blocks in two dimensions (both dimensions equal to the size of selected object list) with threads equal to the size of unselected object list. It identifies the object pairs as mentioned in section 3.2.

$$avoid\_duplicates\_one\_five\_opt1 <<< h\_sol\_len - 1, h\_sol\_len - 1 >>> (d\_sol);$$

launching blocks equal to the size of solution generated by above kernel call with equal number of threads. It avoids the duplicates as mentioned in section 3.2 point 4.

$$update\_best\_ant\_one\_five\_opt1\_first <<< 1, h\_m >>> (d\_best\_ant\_so\_far, d\_sol\_new, d\_rr, h\_m);$$

$$update\_best\_ant\_one\_five\_opt1\_rest <<< 1, h\_m, sizeof(unsignedint) * h\_m >>> (d\_best\_ant\_so\_far, d\_sol\_new[i], d\_rr, h\_m);$$

launching 1 block with threads equal to the number of constraints. It updates the actual solution based on the new results obtained by the local search.

3.2 1.5-Opt2

The following four kernels are specific for 1.5-opt2 local search.

$$dim3block\_dim(l2-1, l2-1, 1);$$

$$create\_sol\_one\_five\_opt2 <<< block\_dim, l1 >>> (d\_sol, d\_sol\_len, d\_obj\_list1, d\_obj\_list2, d\_pp, d\_rr, d\_best\_ant\_so\_far, h\_m);$$

launching blocks in two dimensions (both dimensions equal to the size of unselected object list) with threads equal to the size of selected object list. It identifies the object pairs as mentioned in section 3.6.2.

$$avoid\_duplicates\_one\_five\_opt2 <<< h\_sol\_len-1, h\_sol\_len-1 >>> (d\_sol);$$

launching blocks equal to the size of solution generated by above kernel call with equal number of threads. It avoids the duplicates as mentioned in section 3.2 point 4.

$$update\_best\_ant\_one\_five\_opt2\_first <<< 1, h\_m >>> (d\_best\_ant\_so\_far, d\_sol\_new, d\_rr, h\_m);$$

$$update\_best\_ant\_one\_five\_opt2\_rest <<< 1, h\_m, sizeof(unsignedint) * h\_m >>> (d\_best\_ant\_so\_far, d\_sol\_new[i], d\_rr, h\_m);$$

launching 1 block with threads equal to the number of constraints. It updates the actual solution based on the new results obtained by the local search.

3.2.1 2-Opt

The following four kernels are specific for 2-opt local search:

$$dim3block\_dim(l1-1, l2-1, l2-1);$$

$$dim3thread\_dim(l1-1, 1, 1);$$

$$create\_sol\_two\_opt <<< block\_dim, thread\_dim >>> (d\_sol, d\_sol\_len, d\_obj\_list1, d\_obj\_list2, d\_pp, d\_rr, d\_best\_ant\_so\_far, h\_m);$$

launching blocks in three dimensional (size of x dimension equal to the size of selected object list and y, z dimensions equal to the size of unselected object list) with threads equal to the size of selected object list. It identifies the object pairs as mentioned in section 3.6.3.

$$avoid\_duplicates\_two\_opt <<< h\_sol\_len-1, h\_sol\_len-1 >>> (d\_sol);$$

launching blocks equal to the size of solution generated by above kernel call with equal number of threads. It avoids the duplicates as mentioned in section 3.2 point 4.

$$update\_best\_ant\_two\_first <<< 1, h\_m >>> (d\_best\_ant\_so\_far, d\_sol\_new, d\_rr, h\_m);$$

$$update\_best\_ant\_two\_rest <<< 1, h\_m, sizeof(unsignedint) * h\_m >>> (d\_best\_ant\_so\_far, d\_sol\_new[i], d\_rr, h\_m);$$

launching 1 block with threads equal to the number of constraints. It updates the actual solution based on the new results obtained by the local search.

4 Experimental Results

The experiments are implemented on Intel Xeon E3-1270 v3 CPU and Quadro K2000 GPU. CUDA 5.5 Toolkit was used for all the experiments. The procedure for MKP uses a set of input parameters to find an optimal solution for the heterogeneous environment. A parser module is developed to input the configuration parameters to the program. The input parameters that can be configured are the initial solution, capacity and weight, number of iterations, threshold and learning rate. Several configurations are tested and are shown in the results. For k-exchange algorithms, the configuration is used for finding the optimal solution possible and measuring the speedup with parallelization is shown in Table 2. All the values are plotted in graphs using GraphPad Prism. The gap in the Fig. 2, 3, 4, 6, and 7 graphs is used to reveal the accuracy of the solution value in the specific range.

4.1 Datasets

The MKP instances used are from the OR library, a collection of integer programming problems that contain

MKP instances, among others [28]. There are two MKP problem sets in the OR library, the first contains instances of seven problems. For the second set, the number of dimensions m is 5, 10 and 30, and the number of objects n is 100, 250, and 500. For each combination thirty instances are generated, ten for each tightness factor. The tightness factor of an instance determines the number of objects in the optimal knapsack. The smaller the tightness factor, the less is the number of objects in the optimal knapsack.

4.1.1 Dataset 1

There are seven problems available in dataset 1 from OR library. All seven problems are executed for different combinations of local search algorithms. Since the optimal solution is available for all of the problems in dataset1, it has been used for comparing the quality of the solution generated by the heuristic-based approaches and proposed k-exchange algorithms. Considering the problems in data set 1, the optimal solution values generated for the associated algorithms are represented graphically against the eight configurations of the problem execution shown in Table 2. It has been observed that heuristic-based algorithm itself solves the optimal solution for the problem 1 and problem 5 (Fig. 1 and Fig. 5) only, suggesting a scope for improvement using the k-exchange algorithm. For Problem 2, 1-opt algorithm finds the optimal solution, by improving the solution obtained by the heuristic algorithm. For problem 3 and problem 4, local search 1.5-opt1 finds the optimal solution. In case of problem 6 and problem 7, a significant improvement is observed in the solution with different configurations of local search.

4.1.2 Dataset 2

There are 270 problems available in dataset 2 from OR library. All 270 problems are executed on CPU(serial) as well as on GPU (parallel). Dataset 2 is used for measuring the execution times and speedup of the heuristic and proposed k-exchange algorithms. The configuration list is shown in Table 3. For all of the problems in dataset 2, both serial and parallel execution times of heuristic and four proposed algorithms are measured. The maximum speedup observed for the problem size of 100, 250 and 500 objects is shown in (Fig. 8).

It is shown that the algorithm 2-opt for 500 objects achieves 8.17x speedup for computation whereas algorithm 1.5-opt1 achieves 3.25x speedup. In case of ACO, 1-opt and 1.5-opt1, the speedup is inversely proportional to the increase in the number of objects whereas in 2-opt the speedup is directly proportional to the number of objects. The 1.5-opt2 performance with respect to speedup exhibits unique values and does not fall in the above category. Based on the observations, it is explicitly revealed that GPU speedup is better when 2-opt is considered for 250 and 500 objects whereas the traditional ACO algorithm works well for 100 objects.

Table 2: Configuration for dataset 1

S.No.	Configuration	Configuration Representation in Graph
1	ACO with no local Search	C1
2	ACO with 1-opt	C2
3	ACO with 1.5-opt1	C3
4	ACO with 1.5-opt2	C4
5	ACO with 2-opt	C5
6	ACO with 1-opt then 1.5-opt1	C6
7	ACO with 1-opt, 1.5-opt1 and 1.5-opt2	C7
8	ACO with 1-opt, 1.5opt1, 1.5-opt2 and 2-opt	C8
9	1-opt, 1.5opt1, 1.5-opt2 and 2-opt	C9

Table 3: Configuration for dataset 2

S.No.	Configuration
1	ACO with no local Search
2	Local search 1-opt
3	Local search 1.5-opt1
4	Local search 1.5-opt2
5	Local search 2-opt

Table 4: GPU execution time & optimized solution for TSPLIB instances

Problem	GPU Exec. Time(μ s)		Optimized Solution	
	[29]	Proposed	[29]	Proposed
kroE100	82	76	23025	22382
ch130	82	78	7041	6827
ch150	84	80	7120	6924
kroA200	85	81	31685	30645
ts225	85	82	128513	127659
pr299	87	84	54895	52887
pr439	93	87	115490	110862
pr2392	363	332	412085	390241
pcb3038	547	514	147690	142563
fnl4461	815	772	194746	190492

4.1.3 Dataset 3

TSPLIB library has instances for Symmetric Traveling Salesman Problem (TSP) and Asymmetry Traveling

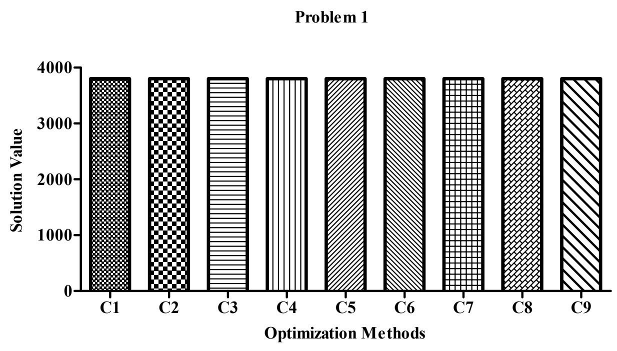

Fig. 1: Solution graph for problem1 (objects - 6 and constraints - 10)

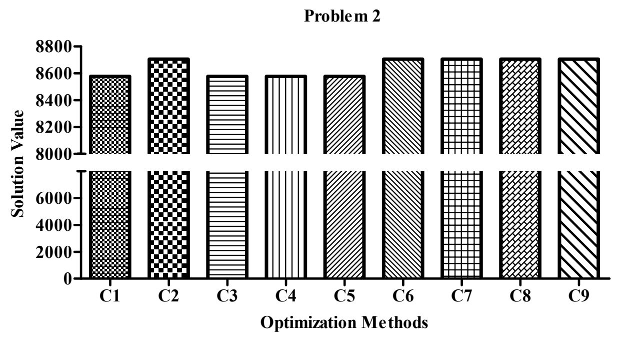

Fig. 2: problem2 (objects - 10 and constraints - 10)

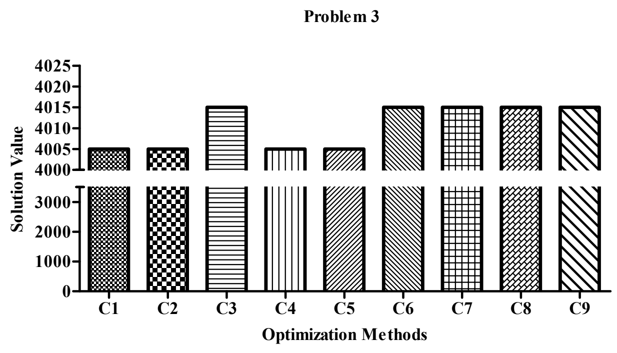

Fig. 3: problem3 (objects - 15 and constraints - 10)

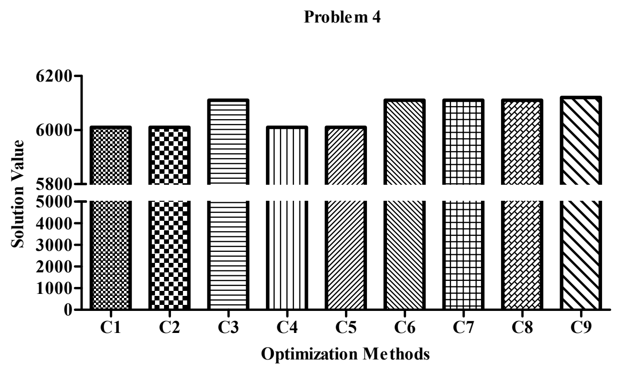

Fig. 4: problem4 (objects - 20 and constraints - 10)

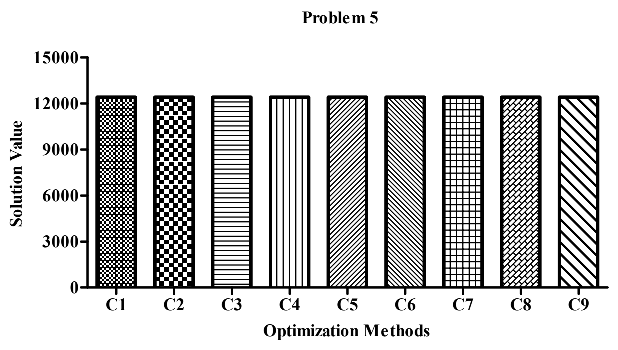

Fig. 5: problem5 (objects-28 and constraints - 10)

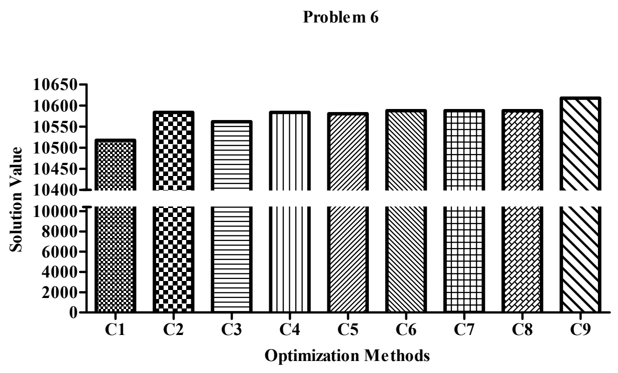

Fig. 6: problem6 (objects - 39 and constraints - 5)

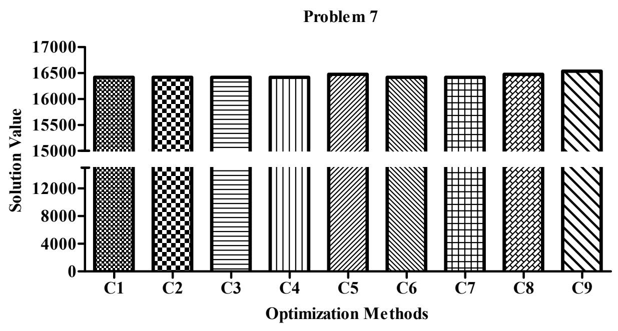

Fig. 7: problem7 (objects - 50 and constraints - 5)

Salesman Problem(ATSP). A set of instances from TSP is used to show the performance [30]. For the opt-2 exchange, there is a significant improvement in terms of GPU execution time and optimal solution as shown in Table 4.

5 Conclusion

Multidimensional knapsack problem and traveling salesman problem instances exhibit irregular memory access pattern. Many data-intensive applications cause memory bandwidth bottleneck on modern many-core architectures due to irregular memory access. To address this issue, we have considered four k-exchange

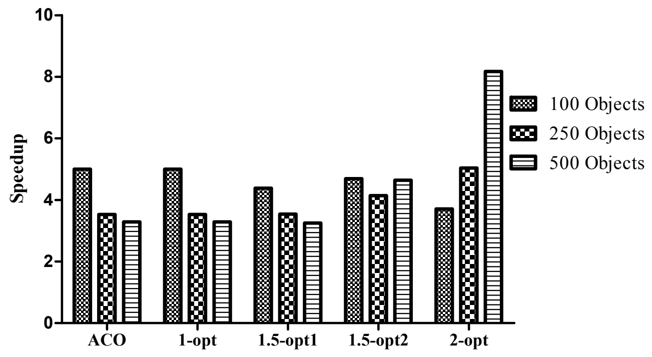

Fig. 8: Maximum speedup observed for problem size of 100, 250 & 500 objects on GPU

algorithms in the transformation step to improve the performance. The optimal solution for most of the problems is obtained by the proposed algorithms. The results show that the proposed k-exchange algorithms execute much faster than the heuristic-based algorithm for the problem size of 250 and above. The proposed procedure for finding an optimal solution can be used for the analysis of communication channel utilization in heterogeneous many-core architectures. A maximum speedup of 8.17x with the k-exchange algorithms has been achieved for MKP instances and significant improvement in TSP instances in terms of GPU execution time and optimal solution. Further, it is important to consider the impact of mixed regular and irregular memory access. Bottleneck analysis of mixed regular and irregular applications on multi-GPUs is still open for future work.

Acknowledgement

The authors are grateful to the anonymous referees for a careful checking of the details and for helpful comments that improved this paper.

References

[1] David B. Kirk and Wen-mei W.Hwu, Programming massively parallel processors: a hands-on approach, Morgan kaufmann (2016).
[2] Nicholas Wilt , The cuda handbook: A comprehensive guide to gpu programming, Pearson Education (2013).
[3] Viktor K. Decyk and Tajendra V. Singh, Particle-in-Cell Algorithms for Emerging Computer Architectures, Comput Phys Commun, Vol. 185, No. 3, pp. 708-719 (2014).
[4] Wen-mei Hwu, What is Ahead for Parallel Computing, J Parallel Distr Com, Vol. 74, No. 7, pp. 2574-2581 (2014).
[5] Baoyuan Liu et al, Sparse convolutional neural networks, Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 806-814 (2015).
[6] Wei Wen, Learning structured sparsity in deep neural networks, Advances in Neural Information Processing Systems, pp. 2074-2082, (2016).
[7] Saumay Dublish, Vijay Nagarajan and Nigel Topham, Characterizing memory bottlenecks in GPGPU workloads, IEEE International Symposium on Workload Characterization (IISWC), pp. 1-2, (2016).
[8] Vijaykumar, Nandita et al., A Case for Core-assisted Bottleneck Acceleration in GPUs: Enabling Flexible Data Compression with Assist Warps, ACM Comp Ar , Vol. 43, No.3, pp. 41-53 (2015).
[9] Amit Kumar Singh et al, Mapping on multi/many-core systems: survey of current and emerging trends, ACM Proceedings of the 50th Annual Design Automation Conference, pp. 1 (2013).
[10] Martin Burtscher, A quantitative study of irregular programs on GPUs, IEEE International Symposium on Workload Characterization (IISWC), pp. 141-151 (2012).
[11] Alfredo and Goldman and Denis Trystram, An Efficient Parallel Algorithm for Solving the Knapsack Problem on Hypercubes, J Parallel Distr Com, Vol. 64, No. 11, pp. 1213-1222 (2004).
[12] Jakob Puchinger et al, The Multidimensional Knapsack Problem: Structure and Algorithms, INFORMS J Comput, Vol. 22, No. 2, pp. 250-265 (2010).
[13] Bing Han, Jimmy Leblet and Gwendal Simon, Hard Multidimensional Multiple Choice Knapsack Problems, An Empirical Study, COMPUT OPER RES, Vol. 37, No.1, pp. 172-181 (2010).
[14] T. C. Martins and M. S. G. Tsuzuki, Solving Irregular Rotational Knapsack Problems, IEEE ISDA 2007 Seventh International Conference on Intelligent Systems Design and Applications, pp. 711-716 (2007).
[15] Wang Jiening, Dong Jiankang and Zhang Chunfeng , Implementation of ant colony algorithm based on GPU, IEEE 2009 Sixth International Conference on Computer Graphics, Imaging and Visualization, pp. 50-53 (2009).
[16] Hongtao Bai, Dantong OuYang, Ximing Li, Lili He and Haihong Yu , MAX-MIN ant system on GPU with CUDA, IEEE Fourth International Conference on Innovative Computing, Information and Control, pp. 801-804 (2009).
[17] Akihiro Uchida, Yasuaki Ito, Koji Nakano, An efficient GPU implementation of ant colony optimization for the traveling salesman problem, IEEE Third International Conference on Networking and Computing(ICNC), pp. 94-102 (2012).
[18] Weihang Zhu and James Curry, Parallel ant colony for nonlinear function optimization with graphics hardware acceleration, SMC 2009. IEEE International Conference on Systems, Man and Cybernetics, pp. 1803-1808 (2009).
[19] Jie Fu, Lin Lei and Guohua Zhou, A parallel ant colony optimization algorithm with GPU-acceleration based on all-in-roulette selection, IEEE Third International Workshop on Advanced Computational Intelligence (IWACI), pp. 260-264 (2010).
[20] Laurence Dawson and Iain Stewart, Improving Ant Colony Optimization performance on the GPU using CUDA, IEEE Congress on Evolutionary Computation (CEC), pp. 1901-1908 (2013).
[21] Sourabh Joshi and Sarabjit Kaur, Comparative analysis of two different ant colony algorithm for model of TSP,

IEEE International Conference on Advances in Computer Engineering and Applications (ICACEA), pp. 669-671 (2015).
[22] Laurence Dawson and Iain A. Stewart, Accelerating ant colony optimization-based edge detection on the GPU using CUDA, IEEE Congress on Evolutionary Computation (CEC), pp. 1736-1743 (2014).
[23] Chiranjit Changdar, G.S. Mahapatra and Rajat Kumar Pal, An improved genetic algorithm based approach to solve constrained knapsack problem in fuzzy environment, Expert Syst Appl, Vol. 42, No.4, pp. 2276-2286 (2015).
[24] , Stefka Fidanova, Ant colony optimization for multiple knapsack problem and model bias, Lect Notes Comput Sc, pp. 280-287 (2005).
[25] Zhang, Xiaoxia and Liu, Zhe and Bai, Qiuying, A new hybrid algorithm for the multidimensional knapsack problem, Springer International Conference on Intelligent Computing, pp. 191-198 (2011).
[26] Alberto Cano, Juan Luis Olmo and SebastiáN Ventura, Parallel multi-objective Ant Programming for classification using GPUs, J Parallel Distr Com, Vol. 73, No. 6, pp. 713-728 (2013).
[27] Musa Peker, Baha Sen and Pınar Yıldız Kumru, An efficient solving of the traveling salesman problem: the ant colony system having parameters optimized by the Taguchi method, Turk J Electr Eng Co, Vol. 21, pp. 2015-2036 (2013).
[28] J E Beasley, OR-Library, `http://people.brunel.ac.uk/mastjjb/jeb/info.html`.
[29] Kamil Rocki and Reiji Suda, High performance GPU accelerated local optimization in TSP. 2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum,1788-1796, (2013).
[30] Gerhard Reinelt, TSPLIB-A traveling salesman problem library. ORSA J Comput, Vol. 3, No.4 , pp. 376-384, (1991).

Tamizharasan P. S. is currently pursuing Ph.D. in Computer Science and Engineering, National Institute of Technology, Tiruchirappalli, India. He received his master degree in Computer Science and Engineering from Anna University, Chennai in 2008. His research interests include Digital System Design, Multi/Manycore architecture, Deep Learning and GPU Computing..

Ramasubramanian N. is currently working as Professor in the Department of Computer Science and Engineering at National Institute of Technology, Tiruchirappalli, India. He holds a BE in Electronics and Communications from REC Trichy (1984) and recieved ME in Computer Science & Ph.D. in CS from National Institute of Technology, Tiruchirappalli, India. He worked as a Senior Project Officer in the Department of CSE, Indian Institute of Technology, Madras on Speech Recognition project funded by Govt. of India. His research interests include Parallel Computer Architecture, Multi-Core Architectures, Digital Systems Design, Security Hardware, GPU Computing, SoC Architectures.