Knowledge-Based Systems 299 (2024) 112095

Contents lists available at ScienceDirect

Knowledge-Based Systems

journal homepage: www.elsevier.com/locate/knosys

CUDA-based parallel local search for the set-union knapsack problem

Emrullah Sonuç [a,b,*], Ender Özcan [a]

[a] Computational Optimisation and Learning (COL) Lab, School of Computer Science, University of Nottingham, Nottingham, NG8 1BB, United Kingdom
[b] Department of Computer Engineering, Faculty of Engineering, Karabuk University, Karabük, 78050, Türkiye

ARTICLE INFO

Keywords:
Combinatorial optimisation
Heuristic
Parallel local search
Set-union knapsack problem

ABSTRACT

The Set-Union Knapsack Problem (SUKP) is a complex combinatorial optimisation problem with applications in resource allocation, portfolio selection, and logistics. This paper presents a parallel local search algorithm for solving SUKP on the Compute Unified Device Architecture (CUDA) platform in Graphics Processing Units (GPUs). The proposed method employs a compact algorithm that divides the search space into smaller regions. For diversity, each thread in a GPU block starts the search process from a different location in a region using a different initial solution. Each thread then searches the local optimum by utilising communication between individuals through a crossover operator exploiting the best solution within the GPU block. Through extensive experiments on a set of SUKP benchmark instances ranging in size from small to large, we demonstrate the effectiveness of the proposed algorithm in finding high-quality solutions within comparable time frames. Furthermore, a comparative performance analysis with the current state-of-the-art SUKP algorithms reveals the competitive advantage of the proposed method. The GPU-based parallel local search algorithm using uniform crossover is a valuable addition to the repertoire of algorithms addressing SUKP, highlighting its potential for practical applications in real-world decision-making scenarios.

1. Introduction

The Set-Union Knapsack Problem (SUKP) [1], an extension of the 0–1 Knapsack Problem, finds applications in various domains such as flexible manufacturing [1], database partitioning [2], financial decision making [3,4], smart cities [5], and practical scenarios such as applied cryptography [6] and the key pose caching problem [7]. In SUKP, items have profits but no weights. The goal is to detect the items maximising the total profit subject to a capacity constraint. SUKP differs from the 0–1 KP because an item is a subset of elements with weights, and the total weight, calculated by considering each of the elements obtained via uniting all selected items, is not allowed to exceed the maximum knapsack capacity.

$U = \{u_i | i = 1, \ldots, n\}$ represents a set of n elements with corresponding weights in $W = \{w_i > 0 | i = 1, \ldots, n\}$. Similarly, $S = \{U_j | j = 1, \ldots, m\}$ represents a set of m items with corresponding profits in $\mathcal{P} = \{p_j > 0 | j = 1, \ldots, m\}$. Each item is a subset $U_j \subseteq U$. The objective is to find a subset $A \subseteq S$ that maximises profit while ensuring that the sum of the weights of the selected items does not exceed the capacity constraint, C. Formally, SUKP can be written as follows [8]:

$$\max \quad f(A) = \sum_{j \in A} p_j \tag{1}$$

$$\text{s.t.} \quad W(A) = \sum_{i \in \bigcup_{j \in A} U_j} w_i \leq C, \quad A \subseteq S \tag{2}$$

Fig. 1 illustrates a small-scale problem instance of SUKP consisting of four items, six elements and with a knapsack capacity $C = 14$. In this figure, $U = \{u_1, u_2, u_3, u_4, u_5, u_6\}$, $w = \{3, 2, 4, 3, 5, 1\}$, $S = \{U_1, U_2, U_3, U_4\}$ and $p = \{12, 7, 13, 9\}$. Each item is a subset of the element set, so U_1 contains two elements which are u_1 and u_4, $U_1 = \{u_1, u_4\}$. For other items, $U_2 = \{u_1, u_2, u_6\}$, $U_3 = \{u_2, u_5\}$, $U_4 = \{u_3, u_4, u_6\}$. The optimal solution for this problem is $A = \{U_1, U_2, U_3\}$, where the total profit is $p_1 + p_2 + p_3 = 32$ and total weight of the elements u_1, u_2, u_4, u_5 and u_6 is 14. Each element's weight is calculated only once.

SUKP is classified as NP-hard [1], signifying that no known algorithm can solve all instances of the problem in polynomial time. Some exact and approximate algorithms are presented for SUKP, but they are not sufficient to solve NP-hard problems. For this purpose, population-based optimisation algorithms (i.e. metaheuristics) are always attractive to solve this and similar problems in such a way that they can obtain promising solutions in a reasonable time [9]. Recent studies into solving SUKP has focused on designing and improving of various metaheuristics [10,11]. Early approaches explored individual generation techniques, repair operators and transfer functions

* Corresponding author at: Computational Optimisation and Learning (COL) Lab, School of Computer Science, University of Nottingham, Nottingham, NG8 1BB, United Kingdom.
E-mail addresses: emrullah.sonuc@nottingham.ac.uk (E. Sonuç), ender.ozcan@nottingham.ac.uk (E. Özcan).

https://doi.org/10.1016/j.knosys.2024.112095
Received 22 March 2024; Received in revised form 30 May 2024; Accepted 6 June 2024
Available online 8 June 2024

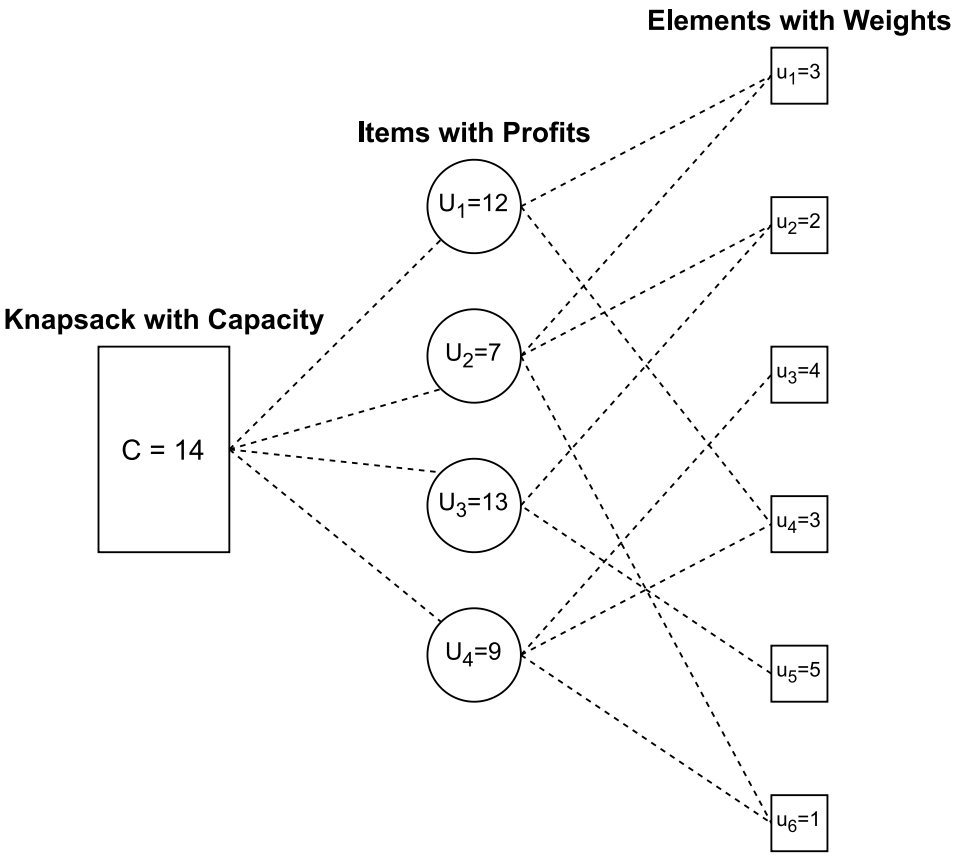

Fig. 1. Illustration of the SUKP.

and obtained promising results. Further investigation has improved these techniques by integrating adaptive features, multiple initialisation strategies and Q-learning frameworks, achieving even better performance and outperforming previous benchmarks [12]. This ongoing effort shows the significant progress made in tackling SUKP and the potential for further development in this area.

To the best of our knowledge, this is one of the initial studies on the use of parallel metaheuristics to solve SUKP. The main contributions of this work are summarised as follows.

First, each thread in a block starts its search from a unique starting point. This strategy is used by all threads simultaneously, ensuring a more comprehensive exploration of the solution space. Starting the search from different points helps to discover unvisited regions during the search process. The threads in each block on the Graphics Processing Unit (GPU) then cooperate to find the local optimum. The search process is more efficient when many threads work together.

Second, since SUKP is a computationally hard constrained optimisation problem, a large amount of computational power is required to improve a given solution or even obtain a feasible solution, hence the need for using the GPU.

Third, the study conducts extensive experiments on a number of SUKP benchmark instances of varying size and complexity. These experiments demonstrate the effectiveness and scalability of the proposed algorithm in efficiently finding high quality solutions within competitive time frames. Furthermore, the comparative performance analysis with existing state-of-the-art SUKP algorithms highlights the competitiveness of the proposed method.

The paper is structured as follows: In Section 2, we review the state-of-the-art methods for SUKP. Section 3 explains the general framework of our proposed approach, including its components. In Section 4, we report the computational results and performance of our approach, along with insightful discussions. Finally, the last section summarises the present study and suggests directions for future work.

2. Related work

Since SUKP is a constrained optimisation problem, it is crucial that how a method handles an infeasible solution. In this context, the related literature handles SUKP with three different approaches. A repair operator converts a infeasible solution into a feasible solution by removing items from the knapsack [8]. A penalty function punishes an infeasible solution by adding a penalty score and makes the search focus on the feasible solution space [13]. Another approach is to search only on the boundary of the feasible solution space and eliminate infeasible solutions directly [14].

He et al. [8] introduced a mathematical model for SUKP and new benchmark problem sets were presented to test the performance of metaheuristics on solving SUKP. This problem set is a popular benchmark that has been frequently used in experiments over the last few years. Genetic algorithm, differential evolution, and artificial bee colony are tested on this problem set and the results demonstrate that the artificial bee colony approach outperforms the other algorithms. Additionally, A-SUKP which is an approximation algorithm based on a greedy strategy proposed by Arulselvan [3] is tested on the problem instances. Feng et al. [15] explored various transfer functions to map solutions from the continuous to the binary domain, achieving improvements in several problem instances compared to previous best-known results. Ozsoydan and Baykasoglu [16] proposed a novel approach to generate a new solution by randomly copying bits from three solutions: the individual's current solution, its most optimal solution thus far, and the best solution among the entire population. This technique is combined with mutation, wherein the probability of mutation is progressively reduced across iterations to encourage exploration. He and Wang [17] proposed a novel repair operator that generates a new solution by incorporating multiple solutions. Lin et al. [13] introduced a penalty function to adaptively explore the boundary of the

search space, intensifying promising solution regions and improving results. Ozsoydan [18] proposed a hybrid approach that employs search characteristics specific to individuals to effectively explore the search space. The probabilities of crossover and mutation are dynamically adjusted to exploit distinct areas of the solution space. Feng et al. [19] enhanced a previous research conducted by Feng et al. [15] by adding a differential mutation and interaction operator to balance between exploration and exploitation. Liu and He [20] introduced a novel repair operator that uses the estimation of distribution method to attain better exploration in the search space. This procedure generates initial solutions through a standard distribution and then employs the Levy flight approach to enhance the solution quality within promising solution regions. Wei and Hao [14] solved six problems optimally using the CPLEX solver. Their method explores neighbouring regions to create a new solution, which increases the probability of finding a local optimum in the search space.

Wu and He [21] presented an improved repair operator, based on the approach proposed by He et al. [8]. The method uses a mutation operator with a Cauchy distribution function for exploration and the crossover mechanism for exploitation.Wei and Hao [22] proposed a method that strategically combines dynamic initialisation with greedy techniques for diversity and tabu search for optimising local search via neighbourhood exploration. Additionally, a non-kernel search strategy generates a new solution randomly to try different areas in the search space. Gölcük and Ozsoydan [23] employed a multi-parent crossover and an adaptive mutation mechanism to achieve an optimal balance between exploration and exploitation. García et al. [24] introduced a novel approach to generate initial solutions using a greedy method. In order to search for solutions in the binary domain, a K-means mapping technique is employed, followed by a local search operator that adds or removes items from the sets to obtain local optima. Durgut et al. [25] investigated how different selection strategies affect the adaptive selection of three binary operators in the search process. Wei and Hao [26] followed Wei and Hao [22] to improve the results using a multi-start strategy. Tabu list manages the solutions in a local region for intensification. Then the proposed method employs a new tabu list to escape from local optima. Results shows a significant performance improvement compared to previous approaches [13,14,21,22]. Dahmani et al. [27] introduced an iterative rounding search-based algorithm utilising partial initial solutions, a local greedy procedure based on profit per potential weight, an exploration method and a diversified strategy by dropping random items. Ozsoydan and Gölcük [28] presented a framework with a Q-learning algorithm serves as a reinforcement method, evaluating optimisation algorithms. To improve the performance of the algorithm, initial solutions are generated based on profit/weight ratios and a random immigrants mechanism is utilised to diversify the population.

Although a method that searches for a feasible solution in the search space achieves more promising results than other methods, it requires many computations and a few parameter adjustments [29]. These methods may be effective because they use a search strategy based on the characteristics of SUKP compared to other methods. While other methods use more general neighbourhood operators, they are also designed for solving other problems except for SUKP. Table 1 summarises an overview of stochastic methods presented for solving SUKP. Many population-based algorithms have been applied to the SUKP. In addition, single-based approaches that incorporate problem-specific knowledge are effective in solving SUKP. It can clearly be observed from the rightmost column of the table, titled "Limits" that it is still needed to develop robust methods and investigate different approaches for handling infeasible solutions.

3. The proposed method

3.1. Main framework: GPU implementation

GPU programming tools have evolved dramatically over the past few years. Compute Unified Device Architecture (CUDA) technology was developed by NVIDIA and has been an essential step in developing GPU computing. This development environment, which is presented with a structure similar to C programming, allows the implementation of parallel algorithms [30,31].

In CUDA architecture, threads are located in blocks. The blocks are in the Grid (see Fig. 2). CUDA executes the instructions on the GPU by calling the *kernel*, which contains a set of functions. The number of threads and blocks in the *kernel* is set on the CPU before calling. In CUDA, each thread block can be synchronised by calling *__syncthreads()* function. All blocks can be synchronised by finalising the *kernel* or with Cooperative Groups offered with CUDA 9.0 [32].

In CUDA technology, memory usage is one of the main factors affecting performance. Each thread has a unique local memory. Each thread block has shared memory with the same lifetime as the block, which is visible to all threads of the block. All threads access the same global memory. CUDA architecture has two different memory spaces called *host* and *device* located on CPU and GPU, respectively. Unified memory provides a managed memory that bridges the memory space of the *host* and *device*. Unified memory is accessible from all CPUs and GPUs in the system as shown in Fig. 3. It allows device memory oversubscription and can significantly simplify the tasks on applications by eliminating the need to copy data on the *host* and *device*.

We use unified memory to design our framework on CUDA, as shown in Fig. 4. Our approach designed as collaborative island model [33], which runs in a CUDA *kernel*. A block represents a subpopulation, i.e. local population, because each thread deals with only one individual in a block [34].

The framework begins by reading the problem instance and configuring the required settings on the host CPU. Then it continues by initialising the GPU device and allocating unified memory to enable seamless data management between the CPU and GPU. The core of the method is a CUDA *kernel* launched on the GPU. Each thread operates independently and generates unique random seeds through the cuRAND library to create initial solutions. The repair operator then checks whether the solutions are feasible. If not, the solution is repaired. A thread within a block generates a candidate solution through the amalgamation of its individual solution with the best solution achieved so far within its specific block using a crossover operator. Applying crossover operator to two feasible solutions might still produce an infeasible new (child) solution. Hence, after crossover, the repair operator is always used to fix any infeasibility that might occur. Iteratively, each thread compares candidate solutions against their current best solution. If the candidate solution is better, it is accepted; otherwise the current solution is retained. This approach encourages collaborative exploration within the search space, with the aim of identifying local optima. This iterative process is synchronised across all threads within a block, ensuring consistency in the search process. After a predefined number of iterations, the algorithm determines the best solution within each block. After this process, *kernel* is finalised. These block-based solutions are then aggregated to determine the overall best solution across the entire GPU.

3.2. Solution representation and handling infeasibility

We use binary encoding to represent a solution to a given SUKP instance. The number of items in a problem instance is denoted by m. Therefore, an m-dimensional binary vector is used as the solution vector, where 1s and 0s represent the selected and unselected items.

SUKP is a constrained optimisation problem that requires a feasibility check when generating a new solution. The new solutions generated by the operators may become infeasible. This can be handled either to by enforcing with penalty functions or by repairing for feasibility with some functions. To repair infeasible solutions or to optimise feasible solutions, we use S-GROA, which was suggested by He et al. [8] and was integrated into many approaches [17,25,28]. This approach is designed to operate in two distinct phases to improve the selection of

Table 1
Summary of the related work.

Reference	Year	Search paradigm	Handling infeasibility	Limits
He et al. [8]	2017	Population-based	Repair operator	Since the method obtains the results in a wide range for most problem instances, it is not effective.
Feng et al. [15]	2018	Population-based	Repair operator	Although best-known values are improved for some problem instances, the results are poor for others.
Ozsoydan and Baykasoglu [16]	2018	Population-based	Repair operator	The method is not robust because it has a wide range between the best and the mean scores compared to other methods.
He and Wang [17]	2018	Population-based	Repair operator	The method is not robust enough with respect to average scores.
Lin et al. [13]	2019	Population-based	Penalty function	The method stucks in local optima for several small-scale instances.
Ozsoydan [18]	2019	Population-based	Repair operator	The method obtains poor results in several trials according to the standard deviations.
Feng et al. [19]	2019	Population-based	Repair operator	The method stucks in local optima for some problems, and its performance is poor in terms of average scores.
Liu and He [20]	2019	Population-based	Repair operator	The method stucks in local optima for some small-scale instances, and its results are poor for large-scale instances.
Wei and Hao [14]	2019	Single-based	Focus only on feasible solutions	Despite a long computational process, it could not reach the best-known values for some small-scale instances.
Wu and He [21]	2019	Population-based	Repair operator	The method is not robust because it has a wide range between the best and the worst scores on several problem instances.
Wei and Hao [22]	2020	Single-based	Focus only on feasible solutions	The method needs more effort to solve SUKP, since three parameters should be configured.
Gölcük and Ozsoydan [23]	2020	Population-based	Repair operator	The method is not robust enough with respect to average scores.
García et al. [24]	2021	Population-based	Repair operator	Although the method achieves promising results, it misses the best-known values in trials, even for small-scale instances.
Durgut et al. [25]	2021	Population-based	Repair operator	The method misses the best-known values in trials, even for small-scale instances.
Wei and Hao [26]	2021	Single-based	Focus only on feasible solutions	The method misses the best-known values in trials, even for small-scale instances.
Dahmani et al. [27]	2022	Single-based	Repair operator	Although the method is effective in solving SUKP, two parameters require fine-tuning.
Ozsoydan and Gölcük [28]	2023	Population-based	Repair operator	The method misses the best-known values for small-scale instances and is not effective for larger ones.

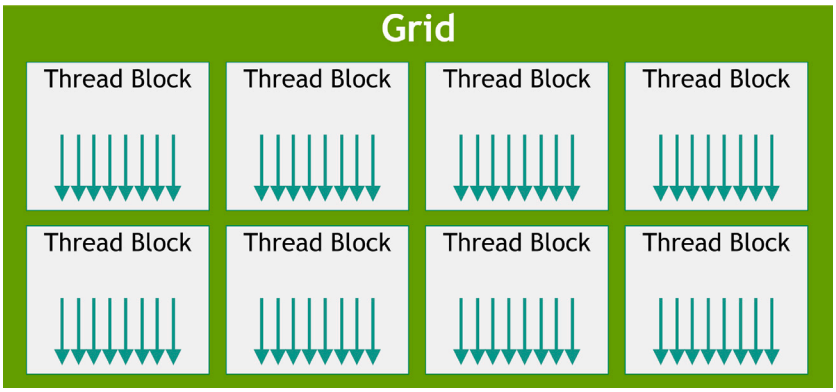

Fig. 2. Grid of thread blocks on CUDA [32].

items for a knapsack. First, it performs a sorting procedure based on the profit and unit resource consumption values assigned to each item. Then, using this ordered data, the algorithm attempts to fix infeasible solutions by eliminating certain items that violate the constraints. Furthermore, after the elimination stage, the method checks whether or not some profitable items can be added to the unused/remaining capacity. Thus, S-GROA tackles the infeasibilities in a solution and attempts to further improve the resultant feasible solution through an additional optimisation process. The steps of S-GROA can more formally be summarised as follows:

1. Calculate the frequency d_j of the element j ($j = 1, 2, \ldots, n$) in the subsets $U_1, U_2, \ldots, U_m$, where U_m is the number of items in which the element u_j is present.
2. Calculate the unit weight $R_i = \sum_{j \in U_i} (w_j / d_j)$ of the item i in U ($i = 1, 2, \ldots, m$).
3. Calculate the profit density $PD_i = p_i / R_i$ of the item i in S ($i = 1, 2, \ldots, m$).
4. Sort all items in S in descending order according to the metric PD, and the index of each item is then stored in a one-dimensional array $H[1...m]$ according to the sorted order.
5. $A_Y = \{i \mid y_i \in Y \text{ and } y_i = 1, 1 \leqslant i \leqslant m\}$ is defined for any binary vector $Y = [y_1, y_2, \ldots, y_m] \in \{0, 1\}^m$.

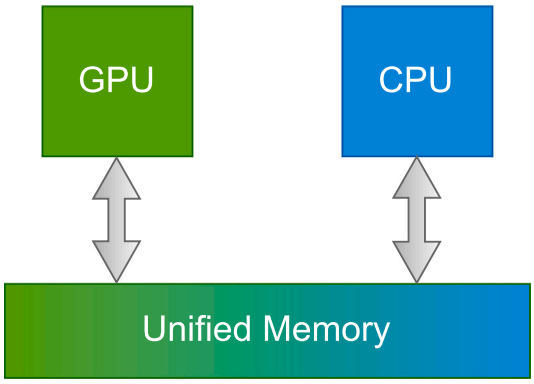

Fig. 3. Illustration of the unified memory.

Using the same instance in Fig. 1, applying S-GROA, the first step yields d_j as $\{2, 2, 1, 2, 1, 2\}$. Following the second and third steps PD is found as $\{4.00, 2.33, 2.17, 1.50\}$. Then the sorted items are placed in the array of H as $[0, 1, 2, 3]$. Suppose that our solution is [0,1,1,1] yielding a total weight of 18 and a profit of 29. This is infeasible since the capacity of the knapsack is exceeded, requiring a repair. Based on array H in S-GROA, the last item in the knapsack is removed and the solution is fixed. After this modification, the resulting solution is [0,1,1,0] with a total weight of 11 and a profit of 20, confirming its feasibility. To improve this solution, S-GROA attempts to include more items. Y = [1,1,1,0] becomes the final feasible solution after this last process with a total weight of 14 and a profit of 32.

3.3. Initial solutions for diversification

In this study, a feasible solution for SUKP is represented by a binary vector with a length of m. Each item in the vector is valued either 1 or 0, indicating if it is in the knapsack or not. We use randomly generated initial solutions to diversify the possible regions. Therefore, each thread starts with a different random solution to explore the search space. A Bernoulli process is followed to create a random number between zero and one, and if it is greater than or equal to 0.5, it is assigned 1, otherwise 0. After completing this process, the threads are ready to generate a candidate solution to exploit their region.

We have used the CUDA random number generation library (cuRAND) to generate random numbers. $curand\_init(thread\_id, 0, 0, \&rndstate)$ initialises the RNG (Random Number Generator) state. $thread\_id$ is used to provide a unique seed for each thread in a CUDA kernel. It helps to ensure that each thread generates different sequences of random numbers. The second and third parameters are used to set the seed for RNG. In this example, both are set to 0, meaning that the seed is generated internally by the cuRAND library based on the $thread\_id$. $\&rndstate$ is a pointer to the $curandState$ variable $rndstate$. After the $curand\_init()$ function is called, $rndstate$ is initialised with the appropriate state for generating random numbers. This state is used by subsequent calls to the cuRAND random number generator functions. S-GROA then carries out the necessary repair and optimisation steps to provide the initial feasible solutions.

3.4. Neighbourhood operator for intensification

To generate a new solution (offspring) for each thread in a block, we apply crossover operator to two 'parent' solutions. Our method outputs only a single new solution instead of two solutions as in the traditional crossovers. A thread in a block (thread X) builds a new solution by applying crossover to

its own solution and the best solution achieved so far within its respective block (thread Y). After applying crossover, the repair operator processes this solution handling infeasibilities if there are any. If the new candidate solution is better than the current one, it is accepted; otherwise rejected and the current solution is kept for the following iteration.

In this study, we have utilised traditional crossover operators of one-point, two-point and uniform crossover operators. In single-point crossover, a randomly chosen point serves as the pivot. Elements before and after this point are swapped between the two solutions. On the other hand, the two-point crossover involves the selection of two different random points, and then the elements between these points are swapped between two solutions. An example illustrating the one-point and two-point crossovers is shown in Fig. 5.

In uniform crossover, if the allele values at a given locus are the same in the parent solutions, it is directly copied into the new solution. If they are not the same, then the allele value of the gene of the thread X is copied into the candidate solution if the uniform random number is less than 0.5, otherwise, the gene of thread Y is copied. An illustration of the uniform crossover operation in a block is shown in Fig. 6.

Table 2
GPU specifications of Nvidia Titan V.

Architecture	Volta
# CUDA cores	5120
# Streaming multiprocessors	80
Clock	1200 MHz
Memory size/type	12 GB/HBM2
Memory bandwidth	652.8 GB/s
Memory bus width	3072 bit
FP32 (float) performance	14.90 TFLOPS

4. Computational experiments

4.1. Experimental environment and SUKP instances

Experiments of the proposed method are carried out on Nvidia Titan V GPU. Table 2 shows the specifications of the GPU used in our experiments. The proposed algorithm is implemented in C++ with the CUDA library (version 11.2).[1] We use cuRAND library to generate pseudo-random numbers. The algorithm seeds that pseudo-random number using the unique thread ID in each thread to ensure diversity across the search region. The proposed algorithm resembles to a genetic algorithm in which we use a population with a large size and carry out one run [35]. We set the number of both blocks and threads to 1024 and the number of iterations to 100 for all tests.

The SUKP instances used in this study are taken from [8], denoted as Set I, and [22], denoted as Set II. In both sets, there are 30 instances categorised into three groups. In the first groups, the number of items is greater than the number of elements, while in the second groups they are equal. In the last groups, the number of items is less than the number of elements. Set I instances contain between 85 and 500 items and elements, while Set II instances contain between 585 and 1000 items and elements. All instances are represented using the notation $m\_n\_\alpha\_\beta$, where m, n, α and β represent the number of items, the number of elements, the density of the elements and the ratio of the knapsack capacity to the total weight of the elements, respectively.

4.2. Performance comparison of crossover operators

We first conducted experiments to evaluate three crossover operators: uniform, one-point, and two-point. The performance of each crossover operator is tested on Set I instances. Table 3 shows the objective function values and the execution time in seconds of the

[1] The source code of the proposed method is available at: https://github.com/3mrullah/CuPLS.

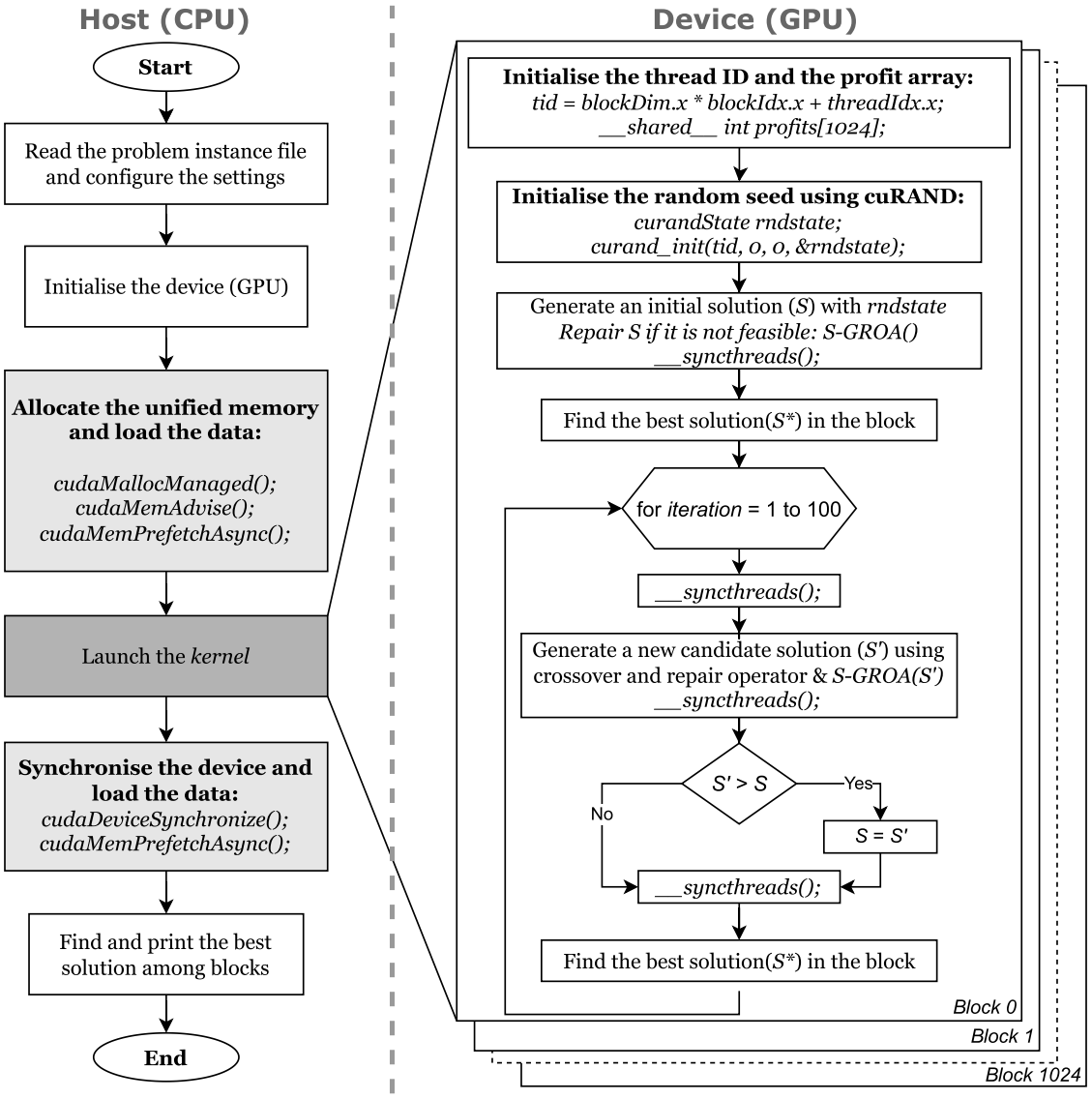

Fig. 4. The framework of our proposed approach.

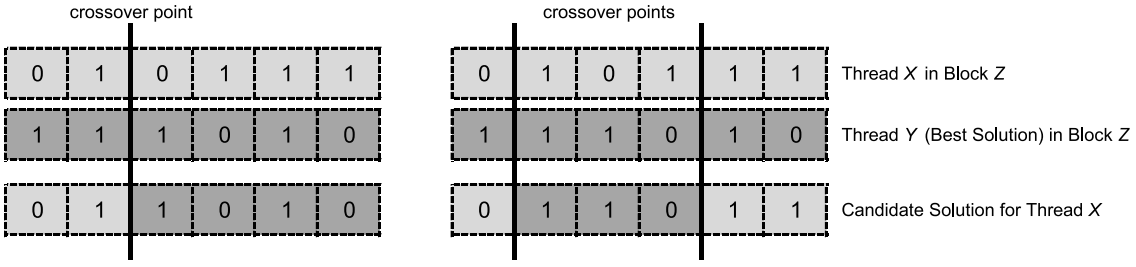

Fig. 5. Graphical illustrations of one-point (on the left) and two-point (on the right) crossovers in a GPU block.

proposed approach with different crossover operators for the first group of instances. The BKS column shows the best-known solutions so far for the problem instances, taken from [26]. For each problem instance, the best value of the crossover in the Obj column is highlighted in bold, and the one that hits the BKS is underlined. It can be seen from the table, the uniform crossover obtained the BKS for instances 100_85_0.10_0.75, 100_85_0.15_0.85, 200_185_0.10_0.75, 200_185_0.15_0.85 and 400_385_0.10_0.75. For the other instances there is a slight difference with the BKS. Two-point crossover has also obtained the BKS for the instances 100_85_0.10_0.75, 100_85_0.15_0.85 and 200_185_0.15_0.85 but shows variation for the others. Unlike the uniform crossover, it cannot obtain the BKS for the instance 200_185_0.10_0.75. The one-point crossover obtained the BKS for only 100_85_0.10_0.75 and 100_85_0.15_0.85. It is the worst method among others since it shows larger deviations for many instances such as 300_285_0.10_0.75 and 500_485_0.15_0.85.

In terms of execution times, the uniform crossover ranged from 10.82 *s* for 100_85_0.10_0.75 to 389.41 *s* for 500_485_0.15_0.85, indicating that the time increases with the complexity of the problem instance. The two-point crossover are consistently higher compared to

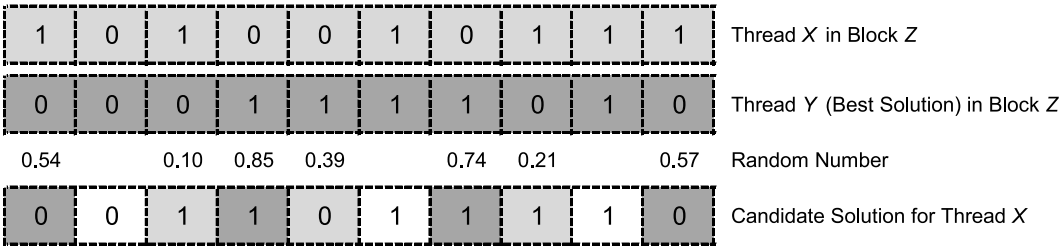

Fig. 6. Graphical illustration of uniform crossover in a GPU block.

Table 3
Computational results of crossover operators on the first group of Set I instances.

Instance	BKS	Uniform		Two-point		One-point	
		Obj	Time (s)	Obj	Time (s)	Obj	Time (s)
100_85_0.10_0.75*	13 283	**13 283**	10.82	**13 283**	14.20	**13 283**	9.80
100_85_0.15_0.85*	12 479	**12 479**	14.11	**12 479**	20.57	**12 479**	12.33
200_185_0.10_0.75	13 521	**13 521**	47.76	13 441	65.54	13 402	54.52
200_185_0.15_0.85	14 215	**14 215**	58.83	**14 215**	81.46	13 949	58.41
300_285_0.10_0.75	11 563	**11 430**	116.41	**11 430**	188.98	11 106	178.43
300_285_0.15_0.85	12 607	**12 500**	139.29	12 402	217.67	12 245	204.01
400_385_0.10_0.75	11 484	**11 484**	209.10	11 443	316.23	11 321	227.27
400_385_0.15_0.85	11 209	**10 757**	245.78	10 756	351.09	10 497	329.92
500_485_0.10_0.75	11 771	**11 722**	329.88	**11 722**	509.10	11 582	504.01
500_485_0.15_0.85	10 238	**10 194**	389.41	10 059	553.42	9613	472.09

Table 4
Computational results of crossover operators on the second group of Set I instances.

Instance	BKS	Uniform		Two-point		One-point	
		Obj	Time (s)	Obj	Time (s)	Obj	Time (s)
100_100_0.10_0.75*	14 044	**14 044**	11.89	**14 044**	18.10	**14 044**	13.87
100_100_0.15_0.85*	13 508	**13 508**	15.17	**13 508**	26.00	**13 508**	21.39
200_200_0.10_0.75	12 522	**12 522**	51.19	**12 522**	79.50	**12 522**	65.44
200_200_0.15_0.85	12 317	**12 317**	64.83	**12 317**	99.43	12 238	80.81
300_300_0.10_0.75	12 817	12 736	115.35	**12 784**	179.00	12 596	146.11
300_300_0.15_0.85	11 585	**11 585**	137.25	11 425	203.48	11 410	193.65
400_400_0.10_0.75	11 665	**11 658**	216.32	**11 658**	333.64	11 211	328.51
400_400_0.15_0.85	11 325	**10 915**	251.31	**10 915**	388.46	10 355	305.51
500_500_0.10_0.75	11 249	**10 924**	341.07	10 921	468.62	10 664	466.70
500_500_0.15_0.85	10 381	**10 381**	391.13	10 194	538.01	9601	497.20

the uniform crossover, ranging from 14.20 s for 100_85_0.10_0.75 to 553.42 s for 500_485_0.15_0.85. The execution times for the one-point crossover are generally lower than the two-point crossover, but higher than the uniform crossover for most instances. Execution times range from 9.80 s for 100_85_0.10_0.75 to 472.09 s for 500_485_0.15_0.85.

The results for a second group of instances (100_100_0.10_0.75 to 500_500_0.15_0.85) are reported in Table 4 and show that the uniform crossover continues to be effective in finding or closely approximating the BKS for most instances, indicating its robustness across different problem instances. Besides, it finds the BKS of the 500_485_0.15_0.85 instance, which is one of the most difficult problems in this group, in less time than other crossover operators. The two-point crossover is less effective at finding the BKS in this group. Furthermore, its execution time indicates that it may not be the best choice for these instances. The one-point crossover has slightly worse performance than the two-point crossover. However, the execution times are generally shorter.

The results for the third group instances are shown in Table 5. It can be seen that the uniform crossover once again shows its effectiveness, offering the best compromise between solution quality and execution time in most instances. The two-point crossover suffers from solution quality problems in many cases and also takes longer, which calls into question its effectiveness for this group. The one-point crossover also suffers from solution quality problems, especially in 385_400_0.10_0.75, 385_400_0.15_0.85 and 485_500_0.15_0.85, and does not offer significant time savings compared to the two-point crossover. The execution times for all crossover operators generally increase with problem complexity, with uniform crossover consistently shorter than the other two methods in this group, as well as in other groups.

Table 6 shows the average rankings calculated by Friedman's test using the objective value for each instance. Uniform crossover is in the first place and two-point crossover is in second place with a slight difference. As a result, the uniform crossover is the most reliable method for all instances on Set I as it consistently achieves the BKS and has the lowest execution times. The two-point crossover does not offer a clear advantage over the uniform crossover in terms of solution quality or time efficiency. Although the one-point crossover occasionally offers slightly better time than the two-point crossover, it does not consistently produce solutions close to the BKS, which may limit its usefulness. The uniform crossover is the most efficient method in terms of both time and solution quality.

4.3. Performance comparison of the proposed approach to sequential algorithms

The comparative results of the proposed method, denoted as CuPLS (CUDA-based parallel local search based on a uniform crossover) from now on, and the state-of-the-art algorithms on the SUKP instances are reported in Tables 7 and 8. The first columns of the tables list the names of each instance, with an asterisk (*) denoting those instances for which CPLEX has verified the optimal value [14]. The performance evaluation of CuPLS is conducted by comparing its mean values with those of state-of-the-art methods, which are shown in individual columns of the table.

Table 5
Computational results of crossover operators on the third group of Set I instances.

Instance	BKS	Uniform		Two-point		One-point	
		Obj	Time (s)	Obj	Time (s)	Obj	Time (s)
85_100_0.10_0.75*	12 045	**12 045**	9.77	**12 045**	17.35	12 020	13.92
85_100_0.15_0.85*	12 369	**12 369**	12.43	**12 369**	21.63	**12 369**	17.27
185_200_0.10_0.75	13 696	**13 696**	46.16	**13 696**	69.27	13 647	56.64
185_200_0.15_0.85	11 298	**11 298**	54.42	**11 298**	76.78	**11 298**	65.98
285_300_0.10_0.75	11 568	**11 568**	110.30	**11 568**	175.84	11 383	169.25
285_300_0.15_0.85	11 802	11 590	131.12	**11 610**	178.13	11 285	146.71
385_400_0.10_0.75	10 600	10 483	206.42	**10 600**	306.73	10 113	287.97
385_400_0.15_0.85	10 506	**10 302**	238.35	**10 302**	366.85	9832	291.52
485_500_0.10_0.75	11 321	**11 260**	329.62	11 070	490.29	10 848	461.95
485_500_0.15_0.85	10 220	**10 117**	385.97	**10 117**	568.07	9608	529.90

Table 6
Average rankings of crossover operators on Set I instances.

Order	Crossover	Rank score
1	Uniform	2.48
2	Two-point	2.28
3	One-point	1.23

Consistent with previous tables, the best value among the methods is indicated in bold within its respective column, while the one that hits the BKS is underlined. Since there is no parallel method proposed for SUKP, we focused the state-of-the-art algorithms: DHJaya [21], HBPSO/TS [13], I2PLS [14], KBTS [22], and MSBTS [26]. DHJaya is a hybrid evolutionary algorithm, HBPSO/TS is a hybrid of particle swarm optimisation and tabu search, I2PLS is an iterated two-phase local search algorithm, KBTS is based on a kernel-based tabu search framework, and MSBTS is a multi-start based tabu search approach. The stopping criterion of these algorithms was 500 s for the Set I instances and 1000 s for the Set II instances and their computational results were taken directly from [26].

CuPLS differs from other serial methods for solving SUKP because of several important advantages. Firstly, CuPLS provides a more efficient parameter tuning procedure as it does not require adjustment of sensitive parameters beyond the number of threads. Furthermore, the determination of the number of threads is facilitated by an evaluation of its performance within the given time constraints. In addition, unlike other algorithms, CuPLS operates only once on a single instance, using an efficient explore-first, exploit-later approach. This strategy increases computational efficiency and simplifies the optimisation process, making CuPLS an efficient solution for tackling SUKP.

After comparing the results on the first group of instances ($m > n$) in Table 7, the following observations can be made: DHJaya and I2PLS achieved BKS in 3 out of 10 instances, CuPLS achieved BKS in 5 out of 10 instances, while HBPSO/TS, KBTS and MSBTS achieved BKS in 6 out of 10 instances. Although HBPSO/TS and MSBTS achieve the BKS value in large-scale instances, they fail to reach it for instance 100_85_0.15_0.85. Similarly, HBPSO/TS, KBTS, and MSBTS also struggle to reach the BKS value for instance 200_185_0.15_0.85. CuPLS has a competitive performance and is equal to the BKS in half of the instances in this group. However, there is some variance in its performance across different instances. For example, CuPLS falls short of BKS on the instances 300_285_0.10_0.75 and 400_385_0.15_0.85, and the scores for these instances indicate that the results are not as consistent as those of HBPSO/TS.

According to the experimental results on the second group of instances ($m = n$), KBTS demonstrates superior performance by achieving the BKS in 7 out of 10 instances. Both HBPSO/TS and CuPLS are the best methods and show similar performance, achieving the BKS in 6 out of 10 instances. While I2PLS misses the BKS for the 100_100_0.15_0.85 instance, it performs well, achieving the highest scores for some of the largest instances in this group (e.g. 300_300_0.10_0.75 and 400_400_0.10_0.75). MSBTS performs poorly on small-scale instances such as 200_200_0.10_0.75 and 200_200_0.15_0.85 and misses the BKS in some trials. DHJaya only performs well on small-scale instances, and tends to fall slightly behind the best performers on large-scale instances (especially at 500_500). Although CuPLS performs poorly on instances 400_400_0.15_0.85 and 500_500_0.10_0.75 instances, its scores are very close to BKS on instances 300_300_0.10_0.75 and 400_400_0.10_0.75. Furthermore, when comparing the results for instance 500_500_0.15_0.85, it is clear that all the methods, except CuPLS, fall short of BKS. Therefore, CuPLS remains highly competitive compared to the other algorithms.

The results on the third group of instances ($m < n$) show that KBTS performs better than the other methods. MSBTS is slightly better than HBPSO/TS and it is the only method that achieves the BKS for instance 485_500_0.10_0.75 in the experiments. While I2PLS and DHJaya have similar overall performance, I2PLS outperforms DHJaya on large-scale instances. HBPSO/TS shows competitive results, especially for medium-sized instances. Its performance drops slightly for larger problem sizes, but it remains effective. CuPLS achieves the BKS for half of the instances in this group. However, instances 285_300_0.15_0.85 and 385_400_0.10_0.75 show slight deviations from the BKS, while instances 385_400_0.15_0.85, 485_500_0.10_0.75 and 485_500_0.15_0.85 show more significant differences. The scores for instances 85_100_0.10_0.75 to 385_400_0.10_0.75 are generally close to the BKS, indicating a consistent level of performance. On the other hand, the scores for instances 385_400_0.15_0.85 to 485_500_0.15_0.85 are significantly lower than the BKS. For this group, the two-point crossover is slightly better than the uniform crossover as it can be seen from the Table 5. However, if the performance of the algorithm falls short between instances 385_400_0.15_0.85 to 485_500_0.15_0.85, it can be improved by improving the crossover mechanism or hybridising it with other methods to increase robustness.

Table 8 shows the results on Set II instances which has large-scale instances than the Set I. MSBTS algorithm outperforms the other algorithms on this set of instances. KBTS and CuPLS have good performance and are very competitive. DHJaya and HBPSO/TS are generally effective on small to medium scale instances such as 600_585_0.15_0.85 and 800_800_0.15_0.85, where they are close to or competitive with the best results. However, as problem size and complexity increase, their performances become poorer; for example, on large-scale and potentially more complex instances such as 900_900_0.10_0.75, it performs significantly worse than other methods. This suggests that while DHJaya and HBPSO/TS are capable of performing on small-scale instances, it is difficult for them to maintain this level of efficiency on large-scale and more complex instances. I2PLS consistently outperforms both DHJaya and HBPSO/TS on a variety of instances. In small-scale instances such as 600_585 and 600_600, I2PLS shows a clear advantage over both DHJaya and HBPSO/TS. However, I2PLS, DHJaya and HBPSO/TS show similar performance on large-scale instances. For example, DHJaya performs better on 1000_985_0.10_0.75 instances, while HBPSO/TS performs better on 1000_1000_0.10_0.75 instances and 1000_1000_0.15_0.85 instances. However, it can be seen

Table 7
A comparison with the reference algorithms on the instances of Set I.

Instance	CuPLS	DHJaya	HBPSO/TS	I2PLS	KBTS	MSBTS
100_85_0.10_0.75*	**13 283**	**13 283**	**13 283**	**13 283**	**13 283**	**13 283**
100_85_0.15_0.85*	**12 479**	**12 479**	12 403.15	12 335.13	**12 479**	12 413.78
200_185_0.10_0.75	**13 521**	13 498.22	**13 521**	**13 521**	**13 521**	**13 521**
200_185_0.15_0.85	**14 215**	**14 215**	14 177.38	14 031.28	14 209.87	13 946.15
300_285_0.10_0.75	11 430	11 167.77	**11 563**	11 562.02	**11 563**	**11 563**
300_285_0.15_0.85	12 500	12 248.42	**12 607**	12 364.55	12 536.02	12 430.51
400_385_0.10_0.75	**11 484**	11 325.88	**11 484**	**11 484**	**11 484**	**11 484**
400_385_0.15_0.85	10 757	10 293.96	**11 209**	11 157.26	**11 209**	**11 209**
500_485_0.10_0.75	11 722	11 675.51	11 746.19	11 729.76	11 755.47	**11 771**
500_485_0.15_0.85	10 194	9703.56	10 163.76	10 133.94	10 202.90	**10 205.62**
100_100_0.10_0.75*	**14 044**	**14 044**	**14 044**	**14 044**	**14 044**	**14 044**
100_100_0.15_0.85*	**13 508**	**13 508**	**13 508**	13 451.50	**13 508**	**13 508**
200_200_0.10_0.75	**12 522**	12 480.62	**12 522**	**12 522**	**12 522**	12 518.28
200_200_0.15_0.85	**12 317**	12 217.81	**12 317**	12 280.07	**12 317**	12 316.21
300_300_0.10_0.75	12 736	12 676.78	12 806.44	**12 817**	**12 817**	12 813.70
300_300_0.15_0.85	**11 585**	11 260.25	**11 585**	11 512.18	11 584.17	**11 585**
400_400_0.10_0.75	11 658	11 301.56	11 484.20	**11 665**	**11 665**	11 657.08
400_400_0.15_0.85	10 915	10 721.45	**11 325**	**11 325**	**11 325**	11 309.20
500_500_0.10_0.75	10 924	10 871.22	11 026.24	11 243.40	11 248.96	**11 249**
500_500_0.15_0.85	**10 381**	10 069.33	10 213.25	10 293.89	10 362.63	10 365.52
85_100_0.10_0.75*	**12 045**	**12 045**	**12 045**	**12 045**	**12 045**	**12 045**
85_100_0.15_0.85*	**12 369**	**12 369**	**12 369**	12 315.53	**12 369**	**12 369**
185_200_0.10_0.75	**13 696**	13 667.63	**13 696**	13 695.60	**13 696**	**13 696**
185_200_0.15_0.85	**11 298**	**11 298**	**11 298**	11 276.17	**11 298**	**11 298**
285_300_0.10_0.75	**11 568**	11 563.80	**11 568**	**11 568**	**11 568**	11 567.70
285_300_0.15_0.85	11 590	11 436.93	**11 802**	11 790.43	11 799.27	11 798.88
385_400_0.10_0.75	10 483	10 287.36	10 552.73	10 536.53	**10 600**	10 599.70
385_400_0.15_0.85	10 302	10 184.09	10 472.40	10 502.64	**10 506**	10 504.23
485_500_0.10_0.75	11 260	10 883.19	11 142.27	11 306.47	11 318.81	**11 321**
485_500_0.15_0.85	10 117	9665.70	10 208.96	10 179.45	**10 219.76**	10 219.04

Table 8
A comparison with the reference algorithms on the instances of Set II.

Instance	CuPLS	DHJaya	HBPSO/TS	I2PLS	KBTS	MSBTS
600_585_0.10_0.75	9699	9449.97	9724.60	9734.74	**9914**	**9914**
600_585_0.15_0.85	9275	8998.45	9174.16	9324.62	9354.52	**9357**
700_685_0.10_0.75	9801	9602	9792.23	9819.24	9844.96	**9881**
700_685_0.15_0.85	9106	8894.09	8940.65	9135.27	9138.36	**9163**
800_785_0.10_0.75	9700	9540.08	9736.89	9678.89	9808.86	**9937**
800_785_0.15_0.85	8803	8649	8872.84	8780.32	8955.29	**8986.25**
900_885_0.10_0.75	9538	9249.53	9560.93	9537.61	9616.70	**9725**
900_885_0.15_0.85	8481	8244.47	8208.22	8426.36	8526.55	**8566.71**
1000_985_0.10_0.75	9485	9306.86	9278.50	9221.23	9496.63	**9632.59**
1000_985_0.15_0.85	8448	8280.52	8129.08	8268.18	8448.05	**8453.36**
600_600_0.10_0.75	10 507	10 504.25	10 517.89	10 520.70	10 521.72	**10 524**
600_600_0.15_0.85	8995	8785.64	8902.33	9022.97	9061.16	**9062**
700_700_0.10_0.75	9648	9409.01	9679.56	9742.73	**9786**	**9786**
700_700_0.15_0.85	9177	8985.51	9003.15	9155.79	9187.55	**9229**
800_800_0.10_0.75	9884	9656.38	9823.17	9685.79	9930.56	**9932**
800_800_0.15_0.85	8907	8774.18	8732.94	8909.50	8936.12	**9101**
900_900_0.10_0.75	9651	9462.86	9639.60	9660.12	9729.51	**9745**
900_900_0.15_0.85	**8990**	8492.88	8617.20	8916	8918.96	**8990**
1000_1000_0.10_0.75	9348	9250.80	9273.64	9255.73	9431.47	**9551**
1000_1000_0.15_0.85	8472	8037.92	7872.84	8206.49	8376.20	**8497.39**
585_600_0.10_0.75	10 304	10 161.45	10 191.01	10 366.15	**10 393**	**10 393**
585_600_0.15_0.85	**9256**	8944.22	**9256**	**9256**	**9256**	**9256**
685_700_0.10_0.75	10 070	9953.55	9909.00	9979.70	10 114.96	**10 121**
685_700_0.15_0.85	9134	8860.79	8936.47	9139.18	**9176**	**9176**
785_800_0.10_0.75	9277	8885.09	9163.90	9236.10	**9384**	**9382.68**
785_800_0.15_0.85	8556	8482.33	8322.17	8558.51	8643.93	**8684.58**
885_900_0.10_0.75	9205	9079.09	9121.24	9106.31	9236.16	**9318**
885_900_0.15_0.85	8217	7881.44	7900.57	8268	8311.68	**8411.72**
985_1000_0.10_0.75	**9234**	8994.48	8938.38	8917.48	9105.84	9193.15
985_1000_0.15_0.85	8528	8425.27	7958.24	8233.05	8488.13	**8578.20**

that DHJaya performs slightly better than I2PLS and HBPSO/TS on large-scale instances. CuPLS consistently achieves competitive results in all instances. It often achieves the highest or close to the highest scores, demonstrating its effectiveness. Specifically, only CuPLS obtained the best value for the one of the largest instance (985_1000_0.10_0.75). This demonstrates the superior performance of CuPLS in handling large-scale and potentially more complex instances, effectively using its parallel processing capabilities to optimise solutions.

As a result on the instances of Set I, CuPLS, unlike the other methods, achieves the BKS for small-scale instances in all groups, indicating a good exploitation capability. Although it achieves values close to the BKS for large-scale instances in this set, it is the only method that achieves the BKS according to the results for 500_500_0.15_0.85, demonstrating the potential and competitiveness of CuPLS. Although CuPLS outperforms KBTS and MSBTS in terms of mean scores for several instances, it is not always able to achieve the BKS. KBTS and

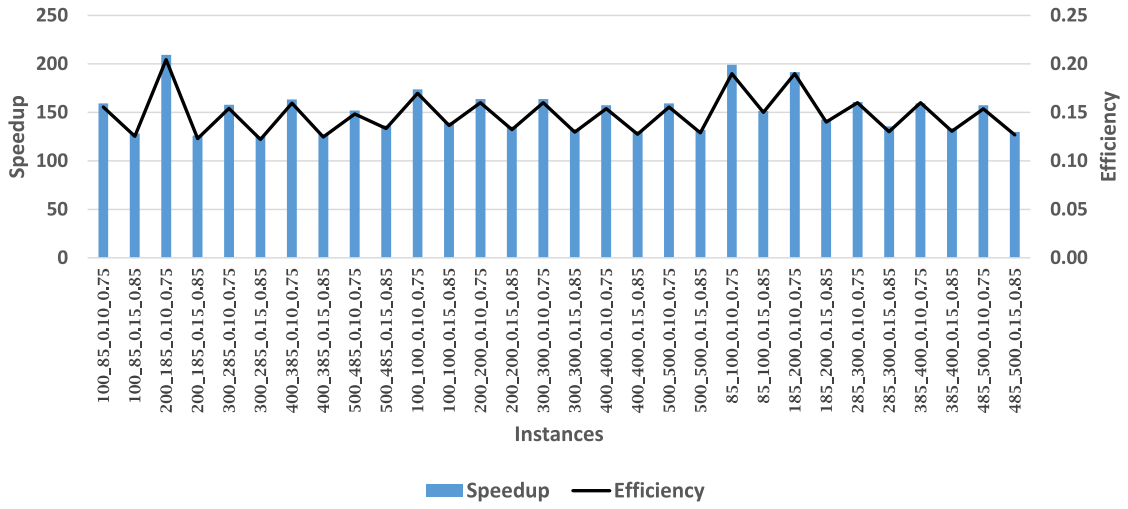

Fig. 7. Speedup and efficiency values for Set I instances.

MSBTS algorithms demonstrate the effectiveness of the tabu search based methods on SUKP. Although these methods perform well on both problem sets, it is observed that MSBTS fails to achieve optimal results on some small-scale instances of Set I, while KBTS shows a worse performance on Set II instances as compared to Set I. It is the fact that only CuPLS can achieve the best values in problem instances such as 200_200_0.15_0.85 and 985_1000_0.10_0.75, and competitive results in other problem instances, indicates that it is an alternative method to SUKP. This suggests that CuPLS has potential and may be more efficient in some instances due to GPU acceleration.

Fig. 7 shows the speedup and efficiency values of CuPLS compared to the sequential implementation running on a single CPU. Each block in CUDA is designed as a population in the serial implementation. The population carries out search on the CPU for a number of iterations equal to the number of blocks times the number of iterations in a block. The computational tests of the method were performed on a computer with a dual-core processor (Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40 GHz), with 16 GB of RAM, running the Windows 10 64-bit operating system. *Speedup* is the ratio of the execution time of the parallel algorithm to that of the sequential algorithm. This measures how much faster the parallel algorithm runs compared to the sequential algorithm. *Efficiency* can be defined as the ratio of speedup and the number of threads executing a parallel algorithm. This measures how well the GPU's parallelism is used. Because the sequential algorithm takes a long time to run on the CPU, it is not efficient and not applicable to SUKP. For this reason, we ran it once to demonstrate the acceleration of CuPLS. Moreover, the experimental results were unsatisfactory as the BKS was not achieved in all instances except 100_100_0.15_0.85. Therefore, due to these limitations, an evaluation based on the objective function value was not performed.

In Fig. 7, the efficiency values range from 0.12 to 0.20 and the speedup values range from 125 to 209. The main reason for various speedup values across different instances of the problem, despite their similar size, may be due to the large number of infeasible solutions generated for these instances. This situation requires frequent use of the repair operator, which affects the overall efficiency and speed of the process. The figure indicates that while our algorithm runs significantly faster on GPU, it seems that we are not exploiting its parallel processing capabilities to the fullest extent possible considering the efficiency indicators for given instances are lower than 1. Synchronising threads and managing their execution can introduce overhead that reduces efficiency. Furthermore, threads within the same block need to execute repair operator, GPU have to serialise these operations, leading to inefficiencies. In summary, although the speedups demonstrate that the GPU offers a significant performance advantage over the CPU for this algorithm, the efficiencies suggest that there is room for optimisation to better utilise the GPU's parallel processing capabilities.

5. Conclusion

This study represents a significant innovation in the field of parallel computational approaches to SUKP. We propose a compact parallel local search algorithm for solving SUKP using high performance computing. By exploiting the parallel processing capabilities of the CUDA architecture, the algorithm achieves remarkable advances in computational efficiency, allowing solution spaces to be explored with unprecedented speed and scalability. Each thread starts the search with different initial solutions. The threads in the blocks then work cooperatively to exploit their region to find the local optima during the iterative search process. In this cooperative stage, the threads generate a candidate solution by using a uniform crossover, amalgamating their respective solutions with the best-solution-found-so-far within their block. During the iterative process, the best solution within each block is maintained. The search process ends by finding the best solution across all blocks. The integration of local search heuristics within a parallel framework represents a novel approach to balancing exploration and exploitation in solution refinement, thereby increasing the effectiveness of the algorithm in achieving near-optimal solutions within a reasonable time frame.

The experimental results from well-known benchmark instances show the effectiveness of the proposed method in detecting high-quality solutions within competitive time frames. Our algorithm achieved competitive performance compared to recent heuristics, making it a viable alternative for SUKP. Furthermore, focusing on acceleration and efficiency, our analysis reveals significant performance gains from our GPU-based algorithm. Not only does it accelerate the search process, but it also enables a thorough exploration of the solution space compared to the sequential CPU implementation.

The CUDA-based parallel local search algorithm shows remarkable effectiveness in solving the SUKP; however, it is crucial to recognise certain inherent limitations. The reliance on local search methods can occasionally lead to suboptimal solutions, especially in cases where the solution space presents irregularities or multimodal characteristics. While our approach attempts to overcome this with multi-threading, the sensitivity of the algorithm to the number of blocks and threads can be a challenge in ensuring consistently superior performance across different problem scenarios.

In summary, our method has certain strengths in terms of consistency and solution quality for the majority of SUKP benchmark instances. The experimental results show that the GPU-accelerated parallel local search algorithm exemplified by our method for solving SUKP shows potential for application to a wider range of problems. Hence, as a trivial future work, we will apply our approach to various

other computationally hard binary-encoded optimisation problems. Our empirical results show that although uniform crossover performs the best in overall, there are cases when two point crossover outperforms uniform crossover on some instances. Hence, the choice of operator is still open for improvement and so studying online and offline intelligent (adaptive) operator selection methods controlling multiple operators [36,37] would be a good future research direction. In our implementation, we have used a generic repair operator since the standard objective function can only be applied to feasible solutions. Another future research direction would be to investigate other objective functions embedding various penalty functions taking the amount of infeasibilities into account to guide the search process. Furthermore, the number of threads or GPU blocks can be dynamically adjusted based on the size or complexity of the problem, potentially improving resource utilisation efficiency and reducing the time required to find optimal solutions.

CRediT authorship contribution statement

Emrullah Sonuç: Writing – original draft, Visualization, Software, Methodology, Investigation, Conceptualization. **Ender Özcan:** Writing – review & editing, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgements

This work was supported by BIDEB-2219 International Postdoctoral Research Fellowship Programme (Grant No. 1059B192001306), thanks to Scientific and Technological Research Council of Türkiye (TÜBİTAK).

References

[1] O. Goldschmidt, D. Nehme, G. Yu, Note: On the set-union Knapsack problem, Naval Res. Logist. 41 (1994) 833–842.
[2] S. Navathe, S. Ceri, G. Wiederhold, J. Dou, Vertical partitioning algorithms for database design, ACM Trans. Database Syst. 9 (1984) 680–710.
[3] A. Arulselvan, A note on the set union Knapsack problem, Discrete Appl. Math. 169 (2014) 214–218.
[4] H. Kellerer, U. Pferschy, D. Pisinger, H. Kellerer, U. Pferschy, D. Pisinger, Knapsack Problems, Springer, 2004.
[5] M. Tu, L. Xiao, System resilience enhancement through modularization for large scale cyber systems, in: 2016 IEEE/CIC International Conference on Communications in China, ICCC Workshops, IEEE, 2016, pp. 1–6.
[6] B. Schneier, Applied Cryptography: Protocols, Algorithms, and Source Code in C, John Wiley & Sons, 2007.
[7] W. Lister, R. Laycock, A. Day, A key-pose caching system for rendering an animated crowd in real-time, Comput. Graph. Forum 29 (2010) 2304–2312.
[8] Y. He, H. Xie, T.-L. Wong, X. Wang, A novel binary artificial bee colony algorithm for the set-union Knapsack problem, Future Gener. Comput. Syst. 78 (2018) 77–86.
[9] I. Dahmani, M. Ferroum, M. Hifi, Effect of backtracking strategy in population-based approach: the case of the set-union Knapsack problem, Cybern. Syst. 53 (2022) 168–185.
[10] I. Dahmani, M. Ferroum, M. Hifi, The local branching as a learning strategy in the evolutionary algorithm: The case of the set-union Knapsack problem, J. Adv. Inf. Technol. 13 (2022).
[11] J. García, A. Leiva-Araos, B. Crawford, R. Soto, H. Pinto, Exploring initialization strategies for metaheuristic optimization: Case study of the set-union Knapsack problem, Mathematics 11 (2023) 2695.
[12] Y. Zhou, M. Zhao, M. Fan, Y. Wang, J. Wang, An efficient local search for large-scale set-union Knapsack problem, Data Technol. Appl. 55 (2021) 233–250.
[13] G. Lin, J. Guan, Z. Li, H. Feng, A hybrid binary particle swarm optimization with tabu search for the set-union Knapsack problem, Expert Syst. Appl. 135 (2019) 201–211.
[14] Z. Wei, J.-K. Hao, Iterated two-phase local search for the set-union Knapsack problem, Future Gener. Comput. Syst. 101 (2019) 1005–1017.
[15] Y. Feng, H. An, X. Gao, The importance of transfer function in solving set-union Knapsack problem based on discrete moth search algorithm, Mathematics 7 (17) (2018).
[16] F.B. Ozsoydan, A. Baykasoglu, A swarm intelligence-based algorithm for the set-union Knapsack problem, Future Gener. Comput. Syst. 93 (2019) 560–569.
[17] Y. He, X. Wang, Group theory-based optimization algorithm for solving Knapsack problems, Knowl.-Based Syst. 219 (2021) 104445.
[18] F.B. Ozsoydan, Artificial search agents with cognitive intelligence for binary optimization problems, Comput. Ind. Eng. 136 (2019) 18–30.
[19] Y. Feng, J.-H. Yi, G.-G. Wang, Enhanced moth search algorithm for the set-union Knapsack problems, IEEE Access 7 (2019) 173774–173785.
[20] X.-J. Liu, Y.-C. He, Estimation of distribution algorithm based on Lévy flight for solving the set-union Knapsack problem, IEEE Access 7 (2019) 132217–132227.
[21] C. Wu, Y. He, Solving the set-union Knapsack problem by a novel hybrid jaya algorithm, Soft Comput. 24 (2020) 1883–1902.
[22] Z. Wei, J.-K. Hao, Kernel based tabu search for the set-union Knapsack problem, Expert Syst. Appl. 165 (2021) 113802.
[23] İ. Gölcük, F.B. Ozsoydan, Evolutionary and adaptive inheritance enhanced grey wolf optimization algorithm for binary domains, Knowl.-Based Syst. 194 (2020) 105586.
[24] J. García, J. Lemus-Romani, F. Altimiras, B. Crawford, R. Soto, M. Becerra-Rozas, P. Moraga, A.P. Becerra, A.P. Fritz, J.-M. Rubio, et al., A binary machine learning cuckoo search algorithm improved by a local search operator for the set-union Knapsack problem, Mathematics 9 (2021) 2611.
[25] R. Durgut, M.E. Aydin, I. Atli, Adaptive operator selection with reinforcement learning, Inform. Sci. 581 (2021) 773–790.
[26] Z. Wei, J.-K. Hao, Multistart solution-based tabu search for the set-union Knapsack problem, Appl. Soft Comput. 105 (2021) 107260.
[27] I. Dahmani, M. Ferroum, M. Hifi, An iterative rounding strategy-based algorithm for the set-union Knapsack problem, Soft Comput. 25 (2021) 13617–13639.
[28] F.B. Ozsoydan, İ. Gölcük, A reinforcement learning based computational intelligence approach for binary optimization problems: The case of the set-union Knapsack problem, Eng. Appl. Artif. Intell. 118 (2023) 105688.
[29] I. Dahmani, M. Ferroum, M. Hifi, S. Sadeghsa, A hybrid swarm optimization-based algorithm for the set-union Knapsack problem, in: 2020 7th International Conference on Control, Decision and Information Technologies, Codit, Vol. 1, IEEE, 2020, pp. 1162–1167.
[30] E. Sonuc, S. Baha, S. Bayir, A parallel simulated annealing algorithm for weapon-target assignment problem, Int. J. Adv. Comput. Sci. Appl. 8 (2017).
[31] E. Sonuc, B. Sen, S. Bayir, A cooperative gpu-based parallel multistart simulated annealing algorithm for quadratic assignment problem, Eng. Sci. Technol. Int. J. 21 (2018) 843–849.
[32] NVIDIA, Cuda c++ programming guide (version 11.8), 2024.
[33] Y. Tan, GPU-Based Parallel Implementation of Swarm Intelligence Algorithms, Morgan Kaufmann, 2016.
[34] A. Munawar, M. Wahib, M. Munetomo, K. Akama, Hybrid of genetic algorithm and local search to solve max-sat problem using nvidia cuda framework, Genet. Program. Evol. Mach. 10 (2009) 391–415.
[35] E. Cantú-Paz, D.E. Goldberg, Are multiple runs of genetic algorithms better than one? in: Genetic and Evolutionary Computation Conference, Springer, 2003, pp. 801–812.
[36] E. Sonuç, E. Özcan, An adaptive parallel evolutionary algorithm for solving the uncapacitated facility location problem, Expert Syst. Appl. 224 (2023) 119956.
[37] J.H. Drake, A. Kheiri, E. Özcan, E.K. Burke, Recent advances in selection hyper-heuristics, European J. Oper. Res. 285 (2020) 405–428.