A *GPU*-based *DP* algorithm for solving multiple instances of the knapsack problem

Dayllon V. X. Lemos, Humberto J. Longo, Wellington S. Martins, Les R. Foulds

[1]Instituto de Informática (INF)
Universidade Federal de Goiás (UFG)
74960-970 – Goiânia – GO – Brasil

dayllonxavier@discente.ufg.br,{longo,wsmartins,lesfoulds}@ufg.br

Abstract. *The knapsack problem is a classic and fundamental optimisation problem that serves as a subproblem in various optimisation algorithms. Thus, it is of great importance that we manage to solve several instances of the knapsack problem in a fast and efficient way. In this work we present a parallel algorithm, based on dynamic programming, that can take advantage of parallelism as more knapsacks need to be solved. The algorithm makes use of fine-grained data parallelism and is easily mapped to GPU accelerators. Extensive experiments with diverse datasets demonstrate the superiority of the proposed algorithm, achieving relevant speedups compared to a serial algorithm.*

Keywords. *Multidimensional Knapsack, Graphics Processing Unit,* MKP*,* GPU*, Dynamic Programming.*

1. Introduction

The *0–1 Multidimensional Knapsack Problem* (*MKP*) belongs to the class $\mathcal{NP}$-hard [Kellerer et al. 2004]. The objective of the MKP, given a set of objects associated with values (gain) and weights (cost), is to select a subset of these objects, maximising the total gain while respecting the capacity limitations in each of its dimensions.

Graphics Processing Units (*GPU*s) are widely recognised and applied in the realm of game development for graphical rendering and image processing. GPUs also facilitate parallel processing in general-purpose applications. This technique is known as *General-Purpose Graphics Processing Unit* (*GP-GPU*), involving the utilisation of GPUs to solve problems commonly unrelated to graphical aspects. Consequently, besides being an efficient means for processing large volumes of data, the GP-GPU can prove effective in resolving substantial instances of optimisation problems, such as the MKP. It can also be beneficial when multiple instances of the same problem need to be repeatedly solved to contribute to the solution of a larger problem.

The MKP has numerous applications in domains such as combinatorial optimisation, cryptography, logistics problems, decision-making, and more. Literature reports showcase diverse applications ranging from capital budgeting and resource allocation [Lorie and Savage 1955], cargo transportation planning [Bellman 1957, Shih 1979], material cutting [Gilmore and Gomory 1966], processor and database allocation in large distributed systems [Gavish and Pirkul 1982], to devising strategies for pollution control and prevention [Bansal and Deep 2012], among many others.

Despite its great usefulness in different domains to model practical problems, even the two-dimensional case of the MKP is generally hard to solve in practice. However, the

need to solve multiple reduced-dimensional sub-instances of an MKP instance (typically with at most three dimensions) can occur as a sub-problem in different MKP resolution approaches. For example, in algorithms that use the *Branch-and-Price* technique to solve an *Integer Linear Programming* (*ILP*) MKP model, this is one of the tasks with the highest processing expense during the search for a viable or optimal solution.

Therefore, in this work we focus on solving the two-dimensional variant of the MKP, denoted here as *KP2*. Firstly, we describe algorithms for both sequential and parallel resolution of the KP2. Section 4.3 formally defines a new problem regarding the simultaneously solving multiple KP2 instances and introduces a parallel approach to solving it using GPU. We propose a dynamic programming-based exact algorithm specifically designed for tackling this problem, which can be readily extended to variants of the problem that encompass more than two dimensions. The algorithm operates on the principle of maximising the workload performed on the GPU and computational experiments conducted using implementations of this algorithm exhibit a significant improvement in efficiency when compared to its sequential counterpart.

The rest of this article is organised as follows. Section 2 outlines some of the works related to solving the MKP using GPUs. Section 3 introduces the notation employed throughout the remainder of the article and formally defines the specific cases addressed. Subsection 4.1 presents a sequential resolution of the KP2. Subsection 4.2 elucidates the transformation of the sequential KP2 resolution algorithm into its fine-grained parallel counterpart. Subsection 4.3 presents strategies for solving multiple instances of the KP2 as a given set. Section 5 is dedicated to the computational results obtained through testing some of the algorithms described in Section 4. Lastly, Section 6 provides final considerations and some prospects for future work.

2. Related Work

Many approaches have been put forth to harness GPU acceleration for solving the MKP. While the majority of the existing literature focuses on exploring parallelism through heuristic-based approximate solutions, there exists a subset of studies that have employed GPU parallelism for exact problem-solving using dynamic programming techniques. In the following section, we highlight a selection of these existing works in the field. However, we did not come across any work that specifically focuses on solving multiple instances of the knapsack problem.

Publications employing GPUs for approximative solutions investigate diverse heuristics within their methodologies. In [Zan and Jaros 2014], the utilisation of the *Particle Swarm Optimization* technique for solving the MKP is discussed. [Fingler et al. 2014] proposes a parallelisation in GPUs of the meta-heuristic *Ant Colony Optimisation*, a technique that uses "artificial ants" to explore possible solution sets for an instance of the MKP. In the study by [de Almeida Dantas and Cáceres 2014], neural networks are employed in addressing the MKP to prevent the best solution from being confined within the vicinity of a local maximum. In [de Almeida Dantas and Cáceres 2015] a resolution method is presented based on the GRASP meta-heuristic (*Greedy Randomised Adaptive Search Procedure*), implemented in parallel. These authors also presented a solution in GPGPU with the probabilistic meta-heuristic *Simulated Annealing* and the GRASP heuristic [de Almeida Dantas and Cáceres 2016, de Almeida Dantas and Cáceres 2018].

A few studies focus on utilising GPUs for the exact solution of the MKP using dynamic programming, and we highlight two of them here. [Berger and Galea 2013] put forward a parallelisation approach for grouping threads in dynamic programming within the CUDA framework. They demonstrate the effectiveness of this approach, particularly in the context of the MKP, where the computation of multi-dimensional indices is resource-intensive. [Biswas and Mukherjee 2022] introduce an implementation of an efficient dynamic programming technique tailored for efficiently solving the Bounded Knapsack Problem (BKP) in a GPU-based system using CUDA. The handling of large-scale problems involves minimal CPU-GPU interactions, optimising memory usage in the GPU for enhanced efficiency. The authors show that this GPU-centric parallel approach exhibits significant speedup compared to an equivalent sequential implementation,

3. Preliminary definitions

The MKP is defined by the following elements: a knapsack of m dimensions; a capacity c_k in the k^{th} knapsack dimension, $k = 1 \dots m$; n objects, where the i^{th} object has a weight $w_{i,k}$ in the k^{th} knapsack dimension, and a profit $p_i, i = 1 \dots n$. The objective of MKP is to select a subset of the n objects with maximum sum of profits, such that the sum of their weights does not exceed any of the capacities c_k, $k = 1 \dots m$.

Let $z_j(c_1, \dots, c_m)$ be the maximum sum of profits that can be feasibly obtained by considering just the first j objects, for some j, where $1 \leqslant j \leqslant n$, while respecting the given fixed capacities $c_1, \dots, c_m$, of the m knapsack dimensions. The optimal value of $z_j(c_1, \dots, c_m)$ is provided by the following recurrence relation:

$$z_j(c_1, \dots, c_m) = \begin{cases} 0, & j = 0; \\ z_{j-1}(c_1, \dots, c_m), & c_k < w_{j,k}, \\ & \quad {\scriptstyle k \in \{1,\dots,m\};} \\ \max \left\{ \begin{array}{l} z_{j-1}(c_1, \dots, c_m), \\ z_{j-1}(c_1 - w_{j,1}, \dots, c_m - w_{j,m}) + p_j \end{array} \right\}, & c_k \geqslant w_{j,k}, \\ & \quad {\scriptstyle k=1,\dots,m.} \end{cases} \tag{1}$$

The relations in (1) are explained as follows. In the base case ($j = 0$) no objects are considered and, trivially, the maximum possible profit is 0. In the second case, it is supposed that the k^{th} dimension capacity c_k of the knapsack for some $k \in \{1, \dots m\}$ is not large enough to contain the weight $w_{j,k}$ of the respective object j. Thus, object j. cannot be added to the current solution subset, which will contain only objects with indices smaller than j. In the final case, it is assumed that all knapsack dimensions accommodate the object j. Therefore, the value of $z_j(c_1, \dots, c_m)$ is the maximum total profit to be obtained with the inclusion, or not, of the object j in the solution subset. In the case of inclusion, the capacities of all dimensions are reduced accordingly.

The optimal value of the recurrence (1), considering all objects, can be obtained by combining the techniques of dynamic programming, using the top-down strategy, and of *memoization* [Pfeffer 2007]. The approach considered in this work was bottom-up, due to the ease of its parallelisation, as described in Subsection 4.2. This approach is characterised by first solving trivial subproblems and then using their solutions to solve more complex subproblems, until the solution of the original problem is reached.

4. KP2 solving approaches

The notation $\&$, used in the algorithms listed in this section, denotes the logical operation *AND* applied bit to bit to two positive integers. A useful task that can be efficiently implemented using this logical operation is the parity check of positive integers, i.e., given a positive integer n, $n \,\&\, 1 = 0$ if n is even, or 1 if it is odd. This expression is used in the following algorithms in order to quickly switch to the entries of the first and the second rows of two-row, multicolumn, matrices, i.e., to toggle between the row of index 0 and the row of index 1. The notation $A[m][n] \leftarrow \{0\}$, denotes both the definition of an array A of dimension $m \times n$ and the initialisation of all its entries with the value 0 (a similar notation was also used for one- and two-dimensional matrices).

The fine-grained parallel algorithms described in this work use the principles of the *CUDA API* (see e.g. [Kirk and Hwu 2016, Nvidia 2023] for more details about GPU and CUDA). In CUDA the data accessed by the kernel (a procedure or code) that is executed in parallel on GPU) must be in the GPU memory and not in the CPU main memory. Thus, before executing a kernel, the necessary data must be copied to the GPU memory and afterwards any useful data must be copied back to the main memory. A program being executed on the CPU can call a kernel subroutine by specifying the number of GPU blocks that will be used and how many threads per block will be launched. The suffix $_{GPU}$ was used for explicit structures existing only in GPU memory, as in the statement $A \leftarrow A_{GPU}$, which indicates the copy of data from GPU memory to the CPU memory.

Parameters $qtyB$ and $qtyT$ of the parallel algorithms described in Subsection 4.2 are, respectively, the number of blocks and the number of threads per block launched in the GPU. The following functions were used to simplify the description of the algorithms: $obtainQtyBlocks()$ returns $qtyB$; $obtainQtyThreadsBlock()$ returns $qtyT$; $obtainThreadId()$ returns the tId identifier of the thread within the block, $0 \leqslant tId < numT$; and $getBlockId()$ returns the block identifier bId, $0 \leqslant bId < qtyB$.

4.1. A sequential algorithm

A sequential solution algorithm for KP2, with asymptotic time and space complexities $\mathcal{O}(n \cdot c_1 \cdot c_2)$, which is strongly based on the recursion (1) and which uses the dynamic programming technique with the bottom-up strategy, is presented in Algorithm 1. Its first phase is the construction of a solution to the subproblem comprising the first n' objects. For this, only the solutions to the subproblems comprising the first $(n' - 1)$ objects are needed (as is evident from the recurrence 1). Thus, both in Algorithm 1 and in the other algorithms presented in this Section, the state matrices M are defined with dimension $2 \times (c_1 + 1) \times (c_2 + 1)$ only and not $n \times (c_1 + 1) \times (c_2 + 1)$. The reason for this is that it is only necessary to alternate between the solutions constructed in the current iteration and in the immediately previous iteration of the algorithms, i.e. it is enough to alternate between only the two rows of M.

Each entry $M[n' \,\&\, 1][c'_1][c'_2]$ stores the value of a solution to the subproblem considering objects with indices less than or equal to n' and capacities c'_1 and c'_2 ($0 \leqslant n' \leqslant n$, $0 \leqslant c'_1 \leqslant c_1$ and $0 \leqslant c'_2 \leqslant c_2$). On the other hand, the entries in the rows $(n' - 1) \,\&\, 1$ of M store the solutions to the subproblems that consider up to $(n' - 1)$ first objects.

However, this particular representation of the M matrix prevents the retrieval of objects from the optimal solution (in the dynamic programming backtracking phase). To

work around this undesirable situation, a second matrix U of binary elements (bits) was used, with dimension $n\times(c_1+1)\times(c_2+1)$, which allows an optimised management of the available memory. Initially, in step 2, all objects are marked as not belonging to the solution subset ($U[i][j_1][j_2]\leftarrow\{0\}$). At the end of the block of steps 3–12, the object i is marked as used ($U[i][j_1][j_2]\leftarrow 1$) if it was added to the solution subset of the subinstance that comprises the first i objects and capacities j_1 and j_2 in the first and second dimensions of the knapsack, respectively.

Algorithm 1: sequentialKP2Solver $(n, c_1, c_2, \boldsymbol{p}, \boldsymbol{w})$

Input: Amount n of objects; capacities c_1 and c_2 of the knapsack dimensions; vector $\boldsymbol{p}$ of profits; matrix $\boldsymbol{w}$ of weights.

Output: Maximum value z. Array Opt of bits (to mark selected objects).

```
1  M[2][c1 + 1][c2 + 1] ← {0};                  // State matrix (2 × (c1 + 1) × (c2 + 1)).
2  U[n][c1 + 1][c2 + 1] ← {0};                    // Bit matrix (n × (c1 + 1) × (c2 + 1)).
3  for i ← 1 to n do
4      for j1 ← 0 to c1 do
5          for j2 ← 0 to c2 do
6              M[i & 1][j1][j2] ← M[(i − 1) & 1][j1][j2];
7              if (j1 ⩾ w_{i,1}  e  j2 ⩾ w_{i,2}) then
8                  c'1 ← j1 − w_{i,1};
9                  c'2 ← j2 − w_{i,2};
10                 if (M[i & 1][j1][j2] < M[(i − 1) & 1][c'1][c'2] + p_i) then
11                     M[i & 1][j1][j2] ← M[(i − 1) & 1][c'1][c'2] + p_i;
12                     U[i][j1][j2] ← 1;
13 z ← M[n & 1][c1][c2];
14 Opt[n] ← {0};                                          // Bit array (n entries).
15 for i ← n to 1 do
16     if (U[i][c1][c2] = 1) then
17         c1 ← c1 − w_{i,1};
18         c2 ← c2 − w_{i,2};
19         Opt[i] ← 1;
20 return z, Opt.
```

When evaluating whether a given object i can be part of the subset Opt of solutions, it is first assumed that i does not belong to such a subset (step 6). Then, it is checked whether or not its addition to the solution subset would exceed any of the capacities of the knapsack (step 7). Finally, if no capacity is exceeded, it is evaluated if the maximum gain already obtained with the first $i-1$ objects (step 10) can be improved. If so, its profit is added to the maximum value obtained so far (step 11). After computing all M matrix states, the maximum value of the solution obtained for the original instance of the problem is available as $M[n][c_1][c_2]$ (step 13). In the particular case where $i=1$, there are subproblems without any objects and the solution value is trivially 0 (defined by the initializing $M[2][c_1+1][c_2+1]\leftarrow\{0\}$ in step 1).

Since it is enough to indicate exactly which objects belong to the Opt solution subset, the subset can be efficiently implemented as a binary $1\times n$ array, as used in the backtracking phase (steps 15 to 19). In this phase all objects are initially marked as not belonging to the Opt subset ($Opt[n]\leftarrow\{0\}$, in step 14). As in step 15 the loop control

variable i is decremented from n to 1, when $U[i][c_1][c_2] = 1$ (step 16) the object i belongs to the Opt subset (step 19). Thus, the weight p_i of the object i is decremented from the remaining capacities c_1 and c_2 of the knapsack (steps 17–18) before starting the next loop iteration. Finally, the z value and Opt array are returned (step 20) by Algorithm 1 as an optimal solution for the input instance.

4.2. Parallel algorithm for the KP2 problem

A proposal for the parallel resolution of instances of the KP2 problem is presented in Algorithms 2.1 and 2.2. The first algorithm runs on the CPU and calls the second algorithm (the kernel), which runs in parallel on a GPU.

Algorithm 2.1: parallelKP2Solver $(n, c_1, c_2, \boldsymbol{p}, \boldsymbol{w}, qtdeB, qtdeT)$

Input: Amount n of objects; capacities c_1 and c_2 of the knapsack dimensions; vector $\boldsymbol{p}$ of profits; matrix $\boldsymbol{w}$ of weights; amounts $qtdeB$ of blocks used on GPU and $qtdeT$ of threads launched in each block.

Output: Maximum value z. Array Opt of bits (to mark selected objects).

```
   // State matrix 2 × ((c1 + 1) · (c2 + 1)) initialised with 0's.
1  M_GPU[2][(c1 + 1) · (c2 + 1)] ← {0};
   // Bit matrix n × (c1 + 1) × (c2 + 1) initialised with 0's.
2  U_GPU[n][c1 + 1][c2 + 1] ← {0};
3  for i ← 1 to n do
4      kernelParallelKP2Solver⟨ qtdeB, qtdeT ⟩(i, c1, c2, p, w, M_GPU, U_GPU);
5  z ← M_GPU[n & 1][c1 · (c2 + 1) + c2];
6  U ← U_GPU;                          // Copy U from GPU memory to CPU memory.
7  Opt[n] ← {0};                       // Array of n bits initialised with 0's.
8  for i ← n to 1 do
9      if (U[i][c1][c2] = 1) then
10         c1 ← c1 − w_{i,1};
11         c2 ← c2 − w_{i,2};
12         Opt[i] ← 1;
13 return z, Opt.
```

Algorithm 2.1 is very similar to Algorithm 1, presented in Section 4.1. The matrices M_{GPU} and U_{GPU} of the former are equivalent to the matrices M and U, respectively, of the latter algorithm. The matrix Opt is identical in both algorithms. These matrices are used in both algorithms for the same purposes. Variable z (step 5 of the Algorithm 2.1) stores the maximum value of the solution to the problem instance and the steps 8 to 12 implement the backtracking phase of the dynamic programming method. Some simple adjustments in Algorithms 2.1 and 2.2, similar to those described in Section 4.1, allow parallel resolution of ***KP1*** as well. Steps 3–4 of Algorithm 2.1 make calls to the kernel (which runs in parallel on the GPU) in order to perform the computation of row i of the matrix M_{GPU} of states. This task is equivalent to steps 4–12 of Algorithm 1.

The two-dimensional matrix M_{GPU} $(2\times((c_1 + 1)\cdot(c_2 + 1)))$ of Algorithm 2.1, has the same role as the three-dimensional matrix M $(2\times(c_1 + 1)\times(c_2 + 1))$ of Algorithm 1. Thus, a state $M_{GPU}[i \,\&\, 1][c'_1 \cdot (c_2 + 1) + c'_2]$ of Algorithm 2.1 is equivalent to state $M[i \,\&\, 1][c'_1][c'_2]$ of Algorithm 1. The decision to keep the matrix M_{GPU} with only two dimensions was made to make easier to implement the kernel.

Algorithm 2.2: kernelParallelKP2Solver $(i, c_1, c_2, \boldsymbol{p}, \boldsymbol{w}, M_{GPU}, U_{GPU})$

Input: Object i to be processed; capacities c_1 and c_2 of knapsack dimensions; vector $\boldsymbol{p}$ of profits; matrix $\boldsymbol{w}$ of weights; state matrix M_{GPU}; matrix U_{GPU} of bits.

```
1  tId ← obtemThreadId() + obtemBlocoId() · obtemQtdeThreadsBloco();
2  passo ← obtemQtdeBlocos() · obtemQtdeThreadsBloco();
3  for j ← tId to (c1 + 1) · (c2 + 1) − 1 by passo do
4      c'1 ← ⌊j/(c2 + 1)⌋;
5      c'2 ← j % (c2 + 1);
6      M_GPU[i & 1][j] ← M_GPU[(i − 1) & 1][j];
7      if (c'1 ⩾ w_{i,1}) e (c'2 ⩾ w_{i,2}) then
8          c''1 ← c'1 − w_{i,1};
9          c''2 ← c'2 − w_{i,2};
10         if (M_GPU[i & 1][j] ⩽ M_GPU[(i − 1) & 1][c''1 · (c2 + 1) + c''2] + p_i) then
11             M_GPU[i & 1][j] ← M_GPU[(i − 1) & 1][c''1 · (c2 + 1) + c''2] + p_i;
12             U_GPU[i][c'1][c'2] ← 1;
```

In the steps 1 and 2 of Algorithm 2.2, respectively, each thread receives an identifier tId, $0 \leqslant tId < qtdeB \cdot qtdeT$ and the numerical variable $step$ is initialised as the number of threads launched on the GPU. After that, at the ℓ-th iteration of the loop contained in steps 3–12, the thread with index *tId* computes entry $M_{GPU}[i][tId + \ell \cdot step]$ of the matrix M_{GPU}. This iterative process runs as long as $tId + \ell \cdot step < (c_1 + 1) \cdot (c_2 + 1)$. Thus, all states of row i of the M_{GPU} matrix are computed, with each thread computing approximately $\frac{(c_1+1)\cdot(c_2+1)}{qtdeB \cdot qtdeT}$ states.

4.3. Parallel resolution of multiple KP2 instances

Let (KP2)^k be the problem of solving simultaneously $k > 1$ KP2 instances that all have the same number of objects, namely, n. The two capacities and the matrices of profits and object weights of the k-th KP2 instance are denoted as c_1^k, c_2^k, $\boldsymbol{p^k}$ and $\boldsymbol{w_1^k}$, respectively. Since solving an instance of the (KP2)^k is equivalent to solving k instances of the KP2, an intuitive parallel approach for solving the (KP2)^k is to run the Algorithm 2.1 with each of the k instances of the KP2, one at a time. However, it is possible to make some adjustments to Algorithm 2.1 in order to generate a higher level of parallelism. This new strategy consists of executing the kernel only once for all the k knapsacks at the same time. That is, the entries of the state matrices for all k knapsacks are computed with just a call to the kernel for each of the objects. This strategy increases the workload of the GPU, but reduces the number of sequential operations in the CPU, consequently requiring less processing time to solve the (KP2)^k instance.

Algorithms 3.1 to 3.3 detail this approach and use a new vector S, where each entry $S[i]$ contains the sum of the knapsack capacities up to the i-th knapsack. Algorithm 3.1 is discussed next. From the defintion of S, $S[i] = \sum_{\ell=1}^{i}(c_1^\ell + 1) \cdot (c_2^\ell + 1)$, for $0 \leqslant i \leqslant k$ (see steps 1–3). Thus, $S[k-1]$ and $S[k] - 1$ are the initial and the final entries, respectively, for the k-th knapsack in the new state matrix. The matrices M_{GPU} and U_{GPU} have their second dimension changed to $S[k]$, which is the size of the concatenation of the rows of the state matrix of all k knapsacks (steps 4 and 5). Step 6 just specifies that the S vector is copied to the GPU memory. Steps 7 and 8 call the kernel, which computes an entire row of the state matrix. The vector Z_{GPU}, which contains the values of the opti-

mal solutions to the k instances KP2, is initially defined in step 9 and updated in step 13. The binary matrix Opt, that represents the subset of objects of each solution, is initially defined in step 10 and updated in step 14.

Algorithm 3.1: multiParallelKP2Solver $(n, k, \boldsymbol{c_1}, \boldsymbol{c_2}, \boldsymbol{p}, \boldsymbol{w}, qtdeB, qtdeT)$

Input: Amount n of objects; amount k of knapsacks; vectors $\boldsymbol{c_1}$ and $\boldsymbol{c_2}$ of capacities; matrix $\boldsymbol{p}$ of profits; matrix $\boldsymbol{w}$ of wights; amount $qtdeB$ and $qtdeT$ of blocks and threads launched in each block on the GPU.

Output: Vector Z of maximum values. Matrix Opt of bits (to mark selected objects).

1 $S[k+1] \leftarrow \{0\}$; `// Integer array with` $(k+1)$ `positions initialised with 0's`
2 **for** $i \leftarrow 1$ **to** k **do**
3 $\quad S[i] \leftarrow S[i-1] + (c_1^i + 1) \cdot (c_2^i + 1)$;
4 $M_{GPU}[2][S[k]] \leftarrow \{0\}$; `// Matrix of` $2 \times S[k]$ `states initialised with 0's.`
5 $U_{GPU}[n][S[k]] \leftarrow \{0\}$; `// Matrix of` $n \times S[k]$ `bits initialised with 0's.`
6 $S_{GPU} \leftarrow S$; `// Copy` S `to GPU memory.`
7 **for** $i \leftarrow 1$ **to** n **do**
8 $\quad$ kernelMultiParallelKP2Solver⟨qtdeB, qtdeT⟩$(k, i, \boldsymbol{c_2}, \boldsymbol{p}, \boldsymbol{w}, M_{GPU}, U_{GPU}, S_{GPU})$;
9 $Z_{GPU}[k] \leftarrow \{0\}$; `// Array with` k `positions initialised with 0's.`
10 $Opt_{GPU}[k][n] \leftarrow \{0\}$; `// Matrix with` $k \times n$ `bits initialised with 0's.`
11 $maxTporB \leftarrow getMaximumNumberOfThreadsPerBlock()$;
12 kernelBacktrackingKP2⟨ $\left\lceil \frac{k}{maxTporB} \right\rceil$, $\min(k, maxTporB)$ ⟩
$\quad (n, k, \boldsymbol{c_1}, \boldsymbol{c_2}, \boldsymbol{w}, M_{GPU}, U_{GPU}, S_{GPU}, Z_{GPU}, Opt_{GPU})$;
13 $Z \leftarrow Z_{GPU}$;
14 $Opt \leftarrow Opt_{GPU}$;
15 **return** Z, Opt.

Algorithm 3.2: kernelMultiParallelKP2Solver $(k, i, \boldsymbol{c_2}, \boldsymbol{p}, \boldsymbol{w}, M_{GPU}, U_{GPU}, S_{GPU})$

Input: Amount k of knapsacks; object i to be processed; vector $\boldsymbol{c_2}$ of capacities; matriz $\boldsymbol{p}$ of profits; matrix $\boldsymbol{w}$ of weights; states matrix M; matrix U of bits; vector S of start positions.

1 $tId \leftarrow obtemThreadId() + obtemBlocoId() \cdot obtemQtdeThreadsBloco()$;
2 $passo \leftarrow obtemQtdeBlocos() \cdot obtemQtdeThreadsBloco()$;
3 $id \leftarrow 1$;
4 **for** $j \leftarrow tId$ **to** $S_{GPU}[k] - 1$ **by** $passo$ **do**
5 $\quad$ **while** $(j \geqslant S_{GPU}[id])$ **do**
6 $\quad\quad id \leftarrow id + 1$;
7 $\quad c'_1 \leftarrow \lfloor (j - S_{GPU}[id-1])/(c_2^{id} + 1) \rfloor$;
8 $\quad c'_2 \leftarrow (j - S_{GPU}[id-1]) \% (c_2^{id} + 1)$;
9 $\quad M_{GPU}[i \,\&\, 1][j] \leftarrow M_{GPU}[(i-1) \,\&\, 1][j]$;
10 $\quad$ **if** $((c'_1 \geqslant w_{i,1}^{id})$ **and** $(c'_2 \geqslant w_{i,2}^{id}))$ **then**
11 $\quad\quad pos \leftarrow S_{GPU}[id-1] + (c'_1 - w_{i,1}^{id}) \cdot (c_2^{id} + 1) + (c'_2 - w_{i,2}^{id})$;
12 $\quad\quad$ **if** $(M_{GPU}[i \,\&\, 1][j] < M_{GPU}[(i-1) \,\&\, 1][pos] + p_i^{id})$ **then**
13 $\quad\quad\quad M_{GPU}[i \,\&\, 1][j] \leftarrow M_{GPU}[(i-1) \,\&\, 1][pos] + p_i^{id}$;
14 $\quad\quad\quad U_{GPU}[i][j] \leftarrow 1$;

A particular feature of the new approach, as exemplified in Algorithm 3.1, is the dynamic programming backtracking phase, which is performed in parallel (step 12). As

the resolution of each KP2 instance implies modifications of different entries in the Z and Opt matrices, independent threads can be launched for each of the k instances. In each of the instances, the value of the solution is retrieved and stored in the Z matrix and the selected objects are indicated by the bits set to 1 in the Opt matrix.

Algorithm 3.3: kernelBacktrackingKP2 $(n, k, \boldsymbol{c_1}, \boldsymbol{c_2}, \boldsymbol{w}, M_{GPU}, U_{GPU}, S_{GPU}, Z_{GPU}, Opt_{GPU})$

Input: Amount n of objects; amount k of knapsacks; vector $\boldsymbol{c_1}$ and $\boldsymbol{c_2}$ of capacities; matrix $\boldsymbol{w}$ of weights of objects; states matrix M_{GPU}; matrix U_{GPU} of bits; vector S_{GPU} of start positions; vector Z_{GPU} of maximum values; matrix Opt_{GPU} of bits.

1 $id \leftarrow 1 + obtemThreadId() + obtemBlocoId() \cdot obtemNumThreadsPorBloco()$;
2 **if** $(id > k)$ **then**
3 **return**;
4 $Z_{GPU}[id] \leftarrow M_{GPU}[n \,\&\, 1][S_{GPU}[id-1] + c_1^{id} \cdot (c_2^{id}+1) + c_2^{id}]$;
5 $c_{aux} \leftarrow c_2^{id} + 1$;
6 **for** $i \leftarrow n$ **to** 1 **do**
7 **if** $(U_{GPU}[i][S[id-1] + c_1^{id} \cdot c_{aux} + c_2^{id}] = 1)$ **then**
8 $c_1^{id} \leftarrow c_1^{id} - w_{i,1}^{id}$;
9 $c_2^{id} \leftarrow c_2^{id} - w_{i,2}^{id}$;
10 $Opt_{GPU}[id][i] \leftarrow 1$;

Given an object i, the kernel for processing a row of the state matrix, is depicted as Algorithm 3.2. Its operation is similar to the other kernel versions described earlier. One difference is that now the variable id specifies the knapsack to which the state being processed belongs. Initially, id is initialised as 1 and, as the control variable j (step 4) is incremented, the new state j may no longer refer to the knapsack id, but to a knapsack of index greater than id. Thus, step 6 does the job of always keeping the id correct with respect to the state j being computed. The remaining steps perform basically the same procedure as in Algorithm 2.2. The main difference is in the use of the expression $(j - S[id-1])$ to calculate the values of the capacities c'_1 and c'_2 (steps 7 and 8). This expression is equivalent to the product $(c_1^{id}+1) \cdot (c_2^{id}+1)$, since $S[id-1]$ specifies the starting entry in the state matrix referring to the id-th instance of the KP2.

The kernel that performs the dynamic programming backtracking process is detailed in Algorithm 3.3. In step 1, the variable id is assigned the index of the instance KP2 relative to thread being run. Steps 2 and 3 end the processing if more threads than the number of instances were launched. In step 4 the capacity of the knapsack id is retrieved through the S vector. The objects are retrieved from steps 6 to 10, in the same way as was in steps 8 to 12 of Algorithm 2.1.

It is worth emphasising that any set of k instances of KP2, even with varying quantities of objects, can be transformed into an equivalent instance of (KP2)^k. To achieve this, it is sufficient to identify the knapsack with the highest number of objects, denoted as n, and for all other knapsack instances with n' objects, where $n' < n$, to introduce $n - n'$ artificial objects. Each of these artificial objects can be defined with zero profits and weights greater than those of the original knapsacks.

5. Comparative tests

This section outlines several computational tests conducted to measure and assess the efficiency of the algorithms presented in this article, particularly concerning process-

ing time. The tests were executed on a computer equipped with an "Intel(R) Xeon(R) CPU @ 2.20GHz" processor and 12 GB of RAM. The utilised GPU was a "Tesla T4" with approximately 15843 MB of memory. The programming languages employed for the implementations were C++ and CUDA, with the GCC 7.5.0 and NVCC 9.2 compilers. The "O3" flag was consistently used as a parameter during compilation.

The tests employed the *MSB* (*M*esyagutov, *S*cheithauer and *B*elov) instance class proposed by [Mesyagutov et al. 2012] from the *2DPackLib* database. This class encompasses instances of the Two-dimensional Orthogonal Packing Feasibility Problem (*2D-OPP*). The instances were modified for the KP2 as follows: the capacities c_1 and c_2 were set equal to the values H and W (height and width of the original plate); the weights $w_{i,1}$ and $w_{i,2}$ and the value p_i of object i were set equal to the height, width, and area of the original instance item, respectively. The optimal solutions obtained for the KP2 instances in this manner do not necessarily correspond to optimal solutions for the respective 2D-OPP instances. However, the values of the solutions for the KP2 instances serve as lower bounds for the optimal values of the corresponding 2D-OPP instances. Henceforth, references to the MSB class pertain to the versions adapted for the KP2.

Table 1. 630 MSB Instances – 20 objects and capacities equal to 1000.

Sequential[1]		Resources[2]		Parallel[3]			Multi Parallel[4]		
Total	**Average**	**Blocks**	**Threads**	**Total**	**Average**	**Speedup**	**Total**	**Average**	**Speedup**
168889,91	268,08	32	32	139682,39	221,71	1,21	10978,94	17,43	15,38
		32	64	133420,01	211,78	1,26	5870,54	9,32	28,76
		64	64	127905,18	203,02	1,32	3358,43	5,33	50,30
		64	128	124914,43	198,28	1,35	2027,44	3,22	83,25
		128	128	125278,91	198,85	1,35	1393,80	2,21	121,30
		128	256	123185,03	195,53	1,37	1100,34	1,75	153,18
		256	256	123837,77	196,57	1,36	1098,56	1,74	154,07

[1] Algorithm 1. [2] Algorithms 2.1–3.3. [3] Algorithms 2.1–2.2. [4] Algorithms 3.1–3.3.

The MSB class contains 630 instances, each with $n = 20$ objects and capacities $c_1 = c_2 = 1000$. Table 1 presents the overall time taken to solve these instances. The first two columns ("Sequential") pertain to the execution of Algorithm 1. The "Total" and "Average" columns display the total and average time taken to solve the 630 instances, respectively. The third and fourth columns ("Resources") indicate the quantities of blocks and threads per block used in the runs of Algorithms 2.1–3.3. The fifth to seventh columns ("Parallel") correspond to solutions produced by the application of Algorithm 2.1. The first two columns in this segment once again, display the total and average time taken to solve the 630 instances, respectively, and the subsequent column presents the average speedup of Algorithm 2.1 over Algorithm 1. Due to the relatively small scale of these instances, the gain from the parallel algorithm is not substantial for this class. The last three columns refer to the simultaneous resolution of all 630 instances by the Algorithm 3.1. The last column describes the average speedup of this algorithm compared to Algorithm 1.

The resolution times of the instances, as presented in Table 1, encompass both processing time, memory allocation and transfer time. Furthermore, for each of the instances, the considered resolution time is the arithmetical mean over 10 runs. The performance gains were superior with 95% statistical significance. It is evident that the decision to

solve all instances at once enabled a higher level of parallelism, augmenting the workload of each thread and reducing the number of kernel calls made by the CPU in comparison to the strategy of addressing one instance at a time. This latter aspect led to the significant efficiency gains through the application of Algorithm 3.1.

6. Conclusion

In Section 4, we introduced algorithms for both sequential and parallel resolution of the KP2 problem (Algorithms 1 and 2.1). Additionally, we have proposed Algorithm 3.1, specifically designed for tackling the extension $(KP2)^k$ outlined in Section 4.3. This algorithm operates on the principle of maximising the workload performed on the GPU.

Section 5 describes the results of computational tests that measured the processing times employed by each algorithm and compared them across the same instances. It was observed that Algorithm 2.1 yielded a substantial efficiency improvement in solving larger instances, when contrasted with Algorithm 1. The results demonstrate that, given the construction of Algorithm 2.1, the greater the capacities of the knapsack dimensions and the fewer the number of objects, the more pronounced this gain becomes. Furthermore, Algorithm 3.1 was devised to concurrently address multiple instances of the KP2 problem, intensifying the workload of each thread dispatched to the GPU. This approach significantly reduced the overall processing time for all instances. It is possible to adapt the structures and processes of the algorithms presented in this study to solve instances of the MKP, regardless of whether they involve one or more dimensions.

During this study, a potential prospect for future research has been discerned. This avenue entails the development of algorithms for solving knapsack problems using the technique of dynamic programming, harnessing the parallel processing capabilities of GPUs, while adopting a top–down approach, which differs from the bottom–up approach presented in the present article. The top–down approach enables the computation of only those subproblems that are directly or indirectly essential for solving the final problem. Although this could potentially result in efficiency gains in terms of execution time and memory when compared to the bottom–up approach, implementing it can be intricate due to its recursive nature. This complexity stems from the recursive nature of the top–down approach. Moreover, the backtracking phase could also become intricate if a flag (bit) was not stored for each subproblem, as is done in the algorithms outlined in this article.

References

Bansal, J. C. and Deep, K. (2012). A Modified Binary Particle Swarm Optimization for Knapsack Problems. *Appl. Math. Comput.*, 218:11042–11061.

Bellman, R. (1957). *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1 edition.

Berger, K.-E. and Galea, F. (2013). An efficient parallelization strategy for dynamic programming on gpu. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, pages 1797–1806. IEEE.

Biswas, G. and Mukherjee, N. (2022). An Efficient Reduced-Memory GPU-based Dynamic Programming Strategy for Bounded Knapsack Problems. In *2022 Seventh In-*

ternational Conference on Parallel, Distributed and Grid Computing (PDGC), pages 18–23. IEEE.

de Almeida Dantas, B. and Cáceres, E. N. (2014). A parallel implementation to the multidimensional knapsack problem using augmented neural networks. In *Proceedings of the 2014 Latin American Computing Conference, CLEI 2014*, pages 1–9.

de Almeida Dantas, B. and Cáceres, E. N. (2015). Sequential and Parallel Implementation of GRASP for the 0-1 Multidimensional Knapsack Problem. In *Procedia Computer Science*, pages 2739–2743.

de Almeida Dantas, B. and Cáceres, E. N. (2016). A Parallelization of a Simulated Annealing Approach for 0-1 Multidimensional Knapsack Problem Using GPGPU. In *Symposium on Computer Architecture and High Performance Computing*, pages 134–140.

de Almeida Dantas, B. and Cáceres, E. N. (2018). An experimental evaluation of a parallel simulated annealing approach for the 0–1 multidimensional knapsack problem. *Journal of Parallel and Distributed Computing*, 120:211–221.

Fingler, H., Cáceres, E., Mongelli, H., and Song, S. (2014). A CUDA based Solution to the Multidimensional Knapsack Problem Using the Ant Colony Optimization. *Procedia Computer Science*, 29:84–94.

Gavish, B. and Pirkul, H. (1982). Allocation of Databases and Processors in a Distributed Computing System. In Akoka, J., editor, *Management of Distributed Data Processing*, volume 31, pages 215–231. North-Holland.

Gilmore, P. C. and Gomory, R. E. (1966). The Theory and Computation of Knapsack Functions. *Operations Research*, 14(6):1045–1074.

Kellerer, H., Pferschy, U., and Pisinger, D. (2004). *Knapsack Problems*. Springer.

Kirk, D. B. and Hwu, W.-M. W. (2016). *Programming massively parallel processors*. Morgan Kaufmann, Oxford, England, 3 edition.

Lorie, J. H. and Savage, L. J. (1955). Three problems in capital rationing. *Journal of Business*, 28(4):229–239.

Mesyagutov, M., Scheithauer, G., and Belov, G. (2012). LP bounds in various constraint programming approaches for orthogonal packing. *Computers & Operations Research*, 39(10):2425–2438.

Nvidia (2023). CUDA C Programming Guide — docs.nvidia.com. `https://docs.nvidia.com/cuda/cuda-c-programming-guide`. [Accessed 09-09-2023].

Pfeffer, A. (2007). Sampling with memoization. In *Proc. of the 22nd National Conference on Artificial Intelligence - Volume 2*, AAAI'07, pages 1263–1270. AAAI Press.

Shih, W. (1979). A Branch and Bound Method for the Multiconstraint Zero-One Knapsack Problem. *Journal of the Operational Research Society*, 30(4):369–378.

Zan, D. and Jaros, J. (2014). Solving the Multidimensional Knapsack Problem using a CUDA accelerated PSO. In *IEEE Congress on Evolutionary Computation, CEC 2014*, pages 2933–2939.