The Journal of Supercomputing
https://doi.org/10.1007/s11227-022-04425-3

Optimization of multi-class 0/1 knapsack problem on GPUs by improving memory access efficiency

En-Ming Huang[1] · **Jerry Chou**[1]

Accepted: 22 February 2022

Abstract
This work aims to improve the GPU performance for solving the 0/1 knapsack problem, which is a well-known combinatorial optimization problem found in many practical applications, including cryptography, financial decision, electronic design automation, computing resource management, etc. The knapsack problem is NP-hard, but it can be solved efficiently by dynamic programming (DP) algorithms in pseudo-polynomial runtime. The DP knapsack algorithm on GPUs has been presented. However, as the modern GPU architecture provides much higher computing throughput than its memory bandwidth, previous work is bounded by the data access time on GPU memory because its CGMA (Compute to Global Memory Access) ratio is 1, which means every computing operation involves one memory access on average. To address the problem, an innovative approach called *Multi-Class 0/1 Knapsack Problem* (MCKP), whose items can be classified into groups with equal values or weights is proposed in this paper. By reconstructing the DP equations for solving MCKP, it is able to explore data parallelism and reusability across threads. This made it possible to optimize the computation across iterations (i.e., items), and significantly improve the CGMA ratio by 5-fold after exploring the use of GPU shared memory and registers for reused data. We extensively analyze the performance of our approach on two modern GPU models, NVIDIA Tesla V100 and RTX 3070. Compared to the runtime of previous work, our approach achieves up to 8x and 18x speedup on V100 and RTX 3070 respectively, the latter one being a GPU with lower memory bandwidth. In addition, by comparing the two speedups, we found that we are able to achieve more efficient computing usage when the memory bandwidth is limited such as RTX 3070.

✉ Jerry Chou
jchou@lsalab.cs.nthu.edu.tw

En-Ming Huang
emhuang@m109.nthu.edu.tw

[1] Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan

Published online: 22 March 2022

Keywords GPU · Parallel computing · Knapsack problem · Performance optimization

1 Introduction

GPUs are highly parallel, many-core architectures that can provide a massive amount of compute throughput compared to the traditional CPU processors. According to the TOP500 list released in June 2021 [8], 6 of the top10 supercomputers in the world are based on GPUs. For instance, the second-ranked machine, Summit [21], has 27,648 NVIDIA Tesla V100 GPU cards installed on 4,608 nodes to provide 148,600 TFLOPS (Tera Floating-Point Operations per Second) performance. The latest announced NVIDIA A100 GPU [19] achieves 19.5 TFLOPS peak single-precision performance and 48.8 GFLOPS per watt energy efficiency. Software stacks like CUDA, which is a parallel computing platform providing API libraries, compiler, runtime, and GPU driver for NVIDIA GPU accelerators, further improve the programmability of GPUs for general-purpose processing by allowing applications to offload their compute-intensive workloads to GPUs for acceleration. Compared to a high-end multi-core CPU, the GPU performance gain is rather one order of magnitude, such as artificial intelligence [38] and scientific computing [9]. Therefore, accelerating applications on GPUs are an active and crucial topic.

In this work, we aim to optimize the acceleration of the famous 0/1 knapsack problem on a single GPU. Given a set of items, each with a weight and a value, a knapsack problem is to choose a subset of items, such that the total value is maximized, while the total weight is less than the capacity of the knapsack. There exist several variants and extensions of the knapsack problem. The most common problem is the 0/1 Knapsack problem, where each item can only be selected at most once and fractional selection of items is not allowable. The knapsack problem is one of the most important combinatorial optimization problems. It has a lot of practical applications, including cryptography [23], financial decision [24], electronic design automation [16], computing resource allocation [10], and power management [12], to name a few. It also commonly appears as a subproblem in analysis and solving of more complicated problems. However, the knapsack problem is known to be NP-hard [6]. Thus several heuristics and approximation schemes, such as firefly algorithm [7] and genetic algorithm [25] have been proposed to find near-optimal solutions in polynomial time. Dynamic programming [1, 31] is also commonly used to solve the problem because it can find the optimal solution in pseudo-polynomial runtime, which is not only polynomial in the numeric value of the number of items but also in the range of data (i.e., the knapsack capacity).

In sum, our paper made the following contributions:

- We formally defined the Multi-Class 0/1 Knapsack Problem (MCKP).
- We investigated and analyzed the performance issue of the MCKP solutions on GPUs.
- We provided the CGMA analysis to prove the problem is memory-bound.

- We proposed a novel approach to optimize the performance by reconstructing the algorithm to reduce computations.
- We proved the correctness of our algorithm reconstruction and analyze the algorithm complexity.
- We evaluate the performance improvement of our algorithms on two latest GPU architecture and discuss the impact of the change of GPU architecture.
- Our work demonstrates the performance of MCKP algorithm can still be improved on GPU. By further improving the computation time of the optimal algorithm, it provides a more attractive option for people to solve the exact MCKP problem in practice.

This paper focuses on optimizing the GPU implementation of the dynamic programming (DP) algorithm for the 0/1 knapsack problem. The DP knapsack algorithm recursively fills in a matrix M, where $M[i][j]$ is a subproblem that finds the maximal value obtained with knapsack capacity j using the first i items. Hence, the optimal solution of a problem with knapsack capacity C and N items can be obtained when $M[N][C]$ is computed. The method to map the DP knapsack algorithm on GPUs is first presented by Boyer et al [3]. In order to match the SIMT (Single Instruction, Multiple Threads) programming model of GPUs and resolve the data dependency among threads, it parallelizes the computations of M in each row as a "*kernel*", which is a group of thread blocks that can be executed in an independent and parallel manner on GPUs. While this method is commonly mentioned by other GPU optimization papers [22, 26], it is found that its performance acceleration is limited on GPU architecture for the three reasons discussed below.

First, launching kernels requires CPU-GPU synchronization and leads to performance deterioration [37]. Boyer's method has to launch a kernel for each row (i.e., item). In other words, n kernels must be launched for a problem instance with n items. Second, Boyer's method is bounded by the memory access bandwidth. According to our analysis, the CGMA (Compute to Global Memory Access) ratio of Boyer's method is only 1, which means every floating-point computing operation requires one access to the *global memory* (i.e., an off-chip memory in GPU). However, the computing throughput of GPUs is much higher than its global memory bandwidth. Lastly, the GPU architecture does offer an on-chip *Shared Memory* installed in each multiprocessor to provide fast data access and avoid global memory access [35]. Furthermore, the data must be copied from global memory in the first place, so only kernels with reused data can benefit from shared memory. Unfortunately, Boyer's method limits the computation of a kernel to a single row, so no reused data exists.

To overcome the aforementioned problems, this work aims to optimize the performance for solving the 0/1 knapsack problem that contains classes of items with the same value (or weight). Such problem instances are commonly seen when the item's values (or weights) are bounded within a range smaller than the number items or when they are divided into a set of classes, such as the application described in [12]. We name such 0/1 knapsack problem as the *Multi-Class 0/1 Knapsack Problem* (abbreviated to **MCKP**) and propose a technique to group the computations across multiple rows (or items) in a single kernel. A new DP equation to solve

such problem is proposed. It enables several optimizations to reduce the number of launched kernels from one per item to one per group and exploit the reused data between rows (i.e., items) to utilize the shared memory. The CGMA ratio of our GPU code is increased from 1 to 5. The experiments are conducted on two modern GPU models show that compared to the runtime of Boyer's method, our approach achieved up to 8x and 18x speedups respectively on Tesla V100 and RTX 3070. Our approach shows good scalability with greater speedup improvements under larger problem scales. Finally, we show that the optimized implementation is able to overcome the memory bottleneck issue and achieve better compute performance on RTX 3070 over V100, where RTX 3070 has higher compute performance but lower memory bandwidth. All these performance results and the metric measurements reported from NVIDIA profilers prove our approach successfully maximized the GPU throughput by minimizing the memory overhead.

The rest of the paper is structured as follows. Section 2 discusses our related work. Section 3 introduces the background and challenges of GPU programming. Section 4 explains the existing CPU-based and GPU-based algorithms for 0/1 knapsack problem. Our proposed solution is described in Sect. 5 and evaluated in Sect. 6. Finally, Sect. 7 concludes the paper.

2 Related work

The knapsack problem is used in a wide range of application domains. In security domain, the Merkle-Hellman knapsack cryptosystems are one of the earliest public key cryptosystems [23]. They use a public set of weights $(a_i, \cdots .a_n)$ to encode a message $(x_i, \cdots .x_n), x_j \in \{0, 1\}$ for $1 \leq j \leq n$ into a encoded message s. Hence, an eavesdropper must solve the intractable knapsack problem to recover the message. Nevertheless, a transformed knapsack problem can be easily solved in polynomial time with a simple greedy algorithm if receivers have a private key (superincreasing sequence). The 0/1 knapsack problem has also been used extensively in the structuring of financial planning problems [24], such as budgeting, where a firm needs to choose among a subset of a portfolio of projects (i.e., item) under a fixed budget constraint (i.e., knapsack capacity). The cost and profit of a project are the weight and value of an item. Scheduling problems in various application domains can also be solved after being mapped to a 0/1 knapsack problem. For instance, Liu et al. [16] use knapsack algorithms to solve a DVS scheduling problem on multi-core embedded systems; Kumaraguruparan et al. [12] uses knapsack algorithms to schedule residential appliances' power usage in a dynamic pricing smart grid system. Last but not least, Kelly [10] shows that the general discrete computational resource allocation problem can also be mapped to a knapsack problem. Many of these mapping problems have the property of restricting their item weight or value in a set of classes and thus match our Multi-Class 0/1 knapsack problem definition. Even if the problems cannot be directly matched, we believe our proposed techniques for performance analysis and optimization can benefit the field for solving these problems as well.

Many heuristic algorithms based on search optimization have been proposed to find near-optimal solutions for the knapsack problem in polynomial time, such as firefly algorithm [7] and genetic algorithm [25]. In particular, many attempts have been made to implement and optimize the branch-and-bound (BB) algorithms [2, 4, 13, 27, 28] on GPUs. The BB algorithms are suitable for GPUs because the traversing paths can be broken into independent subproblems and processed in parallel. However, the main challenge of BB algorithms is to minimize the data transfer time and synchronization between GPUs and CPUs, because the coordination among subproblems must be done by the CPU. Therefore, BB and DP are different type of algorithms and have their own performance optimization challenges. The main advantage of BB over DP is that the time complexity and memory usage of BB does not grow proportionally to the knapsack capacity. Hence, BB algorithms can be more efficient for solving the problem instances with large knapsack capacity. However, search optimization algorithms often require careful parameter tuning to find a better approximation solution in reasonable time constraints. Therefore, both DP and other heuristic algorithms are commonly used and studied in practice.

While the dynamic programming (DP) solution for the knapsack problem has been widely used and applied, only a few works discuss its implementation on different parallel processor architectures, such as GPUs. Ulm and Baker [33] solved 2D knapsack problem on SIMD computers using an associative computing model. Nawaz et al. [18] proposed a general strategy for implementing DP algorithms on FPGA. Lin and Storer [15] improved the divide-and-conquer hypercube algorithm proposed by Lee et al. [14] on a NCUBE hypercube computer. Not until recently, Boyer et al. [3] described the first implementation on GPUs in literature. Boyer's method is also mentioned by O'Connell and Mumford [22] for discussing the general implementation of DP algorithms on GPUs. Boyer et al. [3] also proposed a data compressing strategy to store the result vector of choosing decisions for minimizing the memory bandwidth usage between CPU and GPU. In contrast, our work doesn't consider the result vector and aim to optimize the performance for computing the maximum return value instead. The closest work to our paper is Suri et al. [30], which used similar shared memory and grouping strategy to solve the multiple-choice knapsack problem on GPUs. Nevertheless, the multiple-choice knapsack problem allows an item to be selected multiple times in the solution. Most recently, there are still studies to discuss the parallel knapsack algorithm on CPU [31] and GPU [29], respectively. But none of them attempted to optimize the performance by constructing the DP algorithm like our approach.

Lastly, there are several studies discussing the methods to analyze the GPU performance. Wang and Chu [34] proposed a model to estimate the execution time of GPU kernels with both core and memory frequency scaling. Konstantinidis and Cotronis [11] demonstrate a new model for visual performance insights and performance prediction method for GPU kernels. Roofline Model [36] and instruction Roofline Model for GPU [5] was proposed to provide an intuitive approach to identify performance bottlenecks and guide performance optimization. In this work, we analyze the performance based on CGMA (Compute to Global Memory Access)

ratio as an theoretical method to analyze the performance bottleneck. While it is also possible to analyze the performance of our problem using other measurement methods, additional efforts on code profiling or hardware control may be required. Therefore, it could be our future work to analyze the GPU performance of our problem in more depth and identify other optimization opportunities.

3 Background of GPUs

While our work is based on the NVIDIA GPUs and CUDA programming language, our approach can also be implemented by different programming languages, such as OpenCL or OpenACC, for GPUs from other vendors with similar hardware architecture and programming models. Below, the necessary background of GPUs for introducing our approach is provided. For a complete description of CUDA, we refer readers to the NVIDIA programming guide [20].

GPUs are highly parallel, many-core architectures that can provide a massive amount of computing throughput and higher power efficiency then the CPU processors. However, GPUs were considered as specialized processors for graphic processing tasks until people started developing the software stack to make GPU available for general-purpose computing over the past decade. The most successful and commonly-used software stack is CUDA, a programming platform to enable the implementation of GPGPU (General Purpose GPU) programs on NVIDIA hardware.

CUDA adopts a Single-Instruction, Multiple-Thread (SIMT) programming model to run parallel code on a GPU that is consisted of several multiprocessors. It allows programmers to offload code from CPU, the so-called host, to GPU, the so-called device, by executing kernel functions which launch a group of thread blocks called "*grid*". The *blocks* of a grid are dispatched to idle multiprocessors and thus they must be independent of each other. Only at the end of the kernel execution, the threads across the block can be synchronized. On the contrary, the threads of a block are executed on a multiprocessor as a group of 32 parallel threads called *warp*. Hence, all threads in a warp should follow the same execution path; otherwise, code divergence may lead to poor GPU efficiency. Each thread is indexed by its unique *threadID* and *blockID*, which are often used by programmers to let the threads access different data elements and explore data parallelism.

The memory architecture of GPUs is also hierarchical. Each thread has its own register and local memory space to store private data. Only the data stored in *global memory* can be visible to all the threads in a CUDA program. However, global memory is an off-chip memory space installed outside multiprocessors and its access bandwidth is much lower than the register one. To reduce memory access delay, *shared memory* is installed on each multiprocessor to provide higher access performance than global memory. The data stored in shared memory is also visible to the threads within the same block. Data must be copied from global memory to shared memory before accessing.

4 The 0/1 knapsack problem

The knapsack problem is a well-known NP-hard combinatorial optimization problem that has been used in many practical applications. Although there exist several variants and extensions of the knapsack problem, the 0/1 knapsack problem is the classic and the most known form of the problem. First, the formulation of the 0/1 knapsack problem is given in this section. Then, introducing the dynamic programming (DP) algorithm for solving the problem. Finally, we explain the GPU implementation of the DP algorithm proposed by Boyer et al. in [3] and point out its limitations.

4.1 Problem formulation

The knapsack problem is defined as follows. Given a knapsack with capacity C and a set of N items with each item i having its weight w_i and value v_i, the goal is to determine the number of each item x_i to be selected into the knapsack, such that the total weight does not exceed C and the total value is maximized. The **0/1 knapsack problem** further restricts each item can be selected at most once, so the value of x_i can only be either 0 or 1. The problem can be formulated as the following integer programming problem:

$$\begin{aligned} &\textit{Input} : C = \text{knapsack capacity}; N = \text{number of items}, \\ &\qquad w_i = \text{weight of item} i; v_i = \text{value of item} i. \\ &\text{maximize} \sum_{i=1}^{N} v_i x_i \\ &\text{subject to} \sum_{i=1}^{N} w_i x_i \leq C \\ &\qquad x_i \in \{0,1\}, i \in \{1, \cdots, N\}. \end{aligned} \tag{1}$$

4.2 Dynamic programming

The 0/1 knapsack problem can be solved in pseudo-polynomial time by a dynamic programming (DP) algorithm [1] as follows. Let $dp[i, j]$ be the maximum value that can be attained with weight less than or equal to j using the first i items. The maximum value for a problem instance with N items and C knapsack capacity can be found at $dp[N, C]$ by solving the following dynamic programming recursion with $O(N \times C)$ time complexity:

$$dp[i,j] = \begin{cases} 0, & \text{for } i = 0 \\ dp[i-1,j], & \text{for } i \neq 0, j < w_i \\ \max\{dp[i-1,j], dp[i-1,j-w_i] + v_i\}, & \text{for } i \neq 0, j = w_i, \cdots, C \end{cases} \tag{2}$$

4.3 GPU implementation

The parallel implementation of the above DP algorithm on GPUs is first described by Boyer et al. [3]. As observed by the calculations in Eq. 2, the data dependency only exists between an element $dp[i, j]$ and the elements in its previous row $dp[i-1, j], j \in \{0, \cdots, C\}$. Because there is no data dependency between the elements in the same row (i), it is possible to parallelize the computation of each row i on GPUs by having each GPU thread with threadID j to compute the value of an element $dp[i, j]$. To break the dependency between rows, a global barrier across all the threads must be applied. But there is no GPU instruction for synchronization across multiprocessors, it is necessary to decompose the CUDA program into multiple kernels with one kernel per row. As kernels are serialized on GPUs for execution, the kernel for computing row i will not start before the end of the kernel for computing row $i-1$.

The pseudo-code of the CUDA program is shown in Algorithm 1. The problem input w and v are the set of weights and values for items 1, ..., N, and C is the knapsack capacity. Because only the current row ($dp[i]$) and the previous row ($dp[i-1]$) in the dp matrix are needed for computation, the memory space of the two rows can be reused by exchanging the memory pointer of $dp0$ and $dp1$ after each iteration. Also, the code described by [3] applied Toth's method [32] to eliminate the computations of certain elements in the matrix that never lead to optimal solutions. Toth's method can also be applied to our approach with the same performance improvement. Because it is not relevant to our proposed approach, it is not shown in the pseudo-code for simplicity.

Algorithm 1 Boyer's DP of 0/1 Knapsack Problem on GPUs.

```
1:  CPU process
2:  function MAIN_FUNC(N, C, v, w)
3:      Initialize dp0[j] = dp1[j] = 0, for j = 0, ··· , C
4:      Copy_Mem_From_Host_To_Device(dp0)
5:      Copy_Mem_From_Host_To_Device(dp1)
6:      for i = 1 to N do
7:          GPU_kernel_func(dp0, dp1, w_i, v_i)
8:          swap_mem_pointer(dp0, dp1)
9:      end for
10:     Copy_Mem_From_Device_To_Host(dp0)
11:     Return dp0[C]
12: end function
13: GPU process
14: function GPU_KERNEL_FUNC(dp0, dp1, w_i, v_i)
15:     j = threadID
16:     if j < w_i then
17:         dp1[j] ← dp0[j]
18:     else
19:         dp1[j] ← max{dp0[j], dp0[j − w_i] + v_i}
20:     end if
21: end function
```

The pseudo-code in Algorithm 1 has the following performance drawbacks: First, the number of launched kernels grows proportionally to the number of items. Second, shared memory is not used. Third, the kernel is memory-bound, the

CGMA (Compute to Global Memory Access) ratio is 1 as analyzed in Theorem 1. Our proposed approach aims to address all the above problems to reduce the execution time of the algorithm on GPUs.

Theorem 1 *The CGMA* (*Compute to Global Memory Access*) *ratio of the GPU kernel in Algorithm* 1 *is* 1.

Proof As shown by the GPU computation in Algorithm 1 (line 13-21) , each thread performs a total of 3 computing operations (`add` * 2 and `max`), 2 global memory reads (at $dp0[j]$ and $dp0[j - w_i]$), and 1 global memory write (at $dp1[j]$). Therefore, the CGMA (Compute to Global Memory Access) ratio is 1. □

5 Approach

5.1 Performance challenges

Although GPUs offer massive computing throughput, optimizing application performance on GPUs often encounters the following challenges. We briefly discuss these challenges and summarize how they are addressed in this work.

The data dependency constraints among threads To maximize the throughput of GPUs, thread blocks are scheduled and executed independently on multiprocessors. Therefore, it is the responsibility of programmers to resolve data dependency and avoid race conditions. This is also the main reason why not every application can achieve the ideal performance acceleration on GPUs. In this work, the parallelization strategy for the 0/1 knapsack problem is also limited by this factor. While the traditional GPU implementation can only parallelize computations on each item, this work relaxes data dependency constraints to obtain higher parallelism across multiple items. Furthermore, we reconstruct the calculations in the knapsack algorithm to eliminate concurrent writes and avoid unnecessary synchronization overhead among threads.

The synchronization overhead between host (CPU) and device (GPU) GPUs have much higher computing throughput than CPUs, so GPUs are running asynchronously to CPUs once kernels are launched on GPUs by CPUs. Hence, any synchronous event between host and device can throttle the performance of GPUs. In particular, the synchronization points are often introduced in CUDA program when dealing with the kernel launch event and the memory copy event between host and device. The limited memory bandwidth between host and device presents a serious performance bottleneck for utilizing GPUs, because data is located in CPU memory initially. Extensive efforts have been made to increase the bandwidth through new interconnect technology like NVLink or to overlap data transfer time with kernel execution through asynchronous pipeline strategy like the CUDA Stream. The 0/1 knapsack problem is suitable for GPU offloading because the data transfer time for problem inputs is relatively short compared to the computation time. However, multiple kernels must be launched to enforce a global synchronization barrier between

computing iterations. As a result, the number of launched kernels grows proportionally to the number of items and causes greater performance deterioration. This work addresses these issues by minimizing the number of launched kernels and overlapping the data pre-processing time on CPUs with the kernel execution time on GPUs.

The slow access performance on global memory Once kernels are executed on GPUs, the global memory access time becomes the main performance bottleneck because of the huge performance gap between global memory and computing cores. Taking the latest A100 GPU as an example, its computing throughput for double-precision data is 9.7 TFLOPS, but its global memory access bandwidth is only at 1.6TB/s. That implies the time to access an eight bytes double-precision floating-point data on global memory is enough to perform almost 50 floating-point operations. In other words, the performance of the GPU is likely to be bounded by the memory access time unless the ratio between computing and memory access can reach 50. Therefore, improving the *CGMA (Compute to Global Memory Access)* ratio is always a key indicator for performance optimization. In this work, the CGMA ratio of the original GPU method is improved five-fold after exploiting the benefit of shared memory for storing reused data.

Limited memory space Last but not least, the memory space on GPU devices is limited compared to CPU hosts. The largest global memory size on any GPU model is less than 50GB, and the largest shared memory size shared by a thread block is only 48KB. Therefore, the largest problem instance that can be solved on a GPU is often restricted by the memory space. In this work, the problem size is assumed to fit into the global memory space like all the previous papers [3, 22, 30]. However, the shared memory size per thread can be increased with a smaller block size (i.e., number of threads per block). Hence, this work also allows users to trade the degree of parallelism with the usage of shared memory space to achieve the best overall performance.

5.2 Multi-class 0/1 knapsack problem

This work aims to optimize the GPU acceleration performance of 0/1 knapsack problem with multiple items having the same weight or value. This type of problem instance is commonly seen in the following two scenarios. One is when the weight and value of items are bounded within a range that is relatively small compared to the number of items. The other one is when the weight or value of items can be divided into a set of classes in the problem instance, and the number of classes is relatively small compared to the number of items. For example, [12] uses 0/1 knapsack algorithm to solve a power management problem where the power consumption cost of objects in their problem instance is mapped to the value of items in the knapsack problem instance. Because the power consumption cost is modeled by a discrete piecewise constant function, only a limited number of item values exist in the mapped knapsack problem instance.

A new DP algorithm is proposed called *Multi-Class 0/1 Knapsack Problem* (abbreviated to **MCKP**) to solve the above 0/1 knapsack problem, which N

items are grouped into M ($M \leq N$) classes, and all the items in a group have the same value (or weight).

All the approaches and proofs discussed in this section can be applied to the MCKP problem with classes either on item weight or value. Therefore, the MCKP problem with classes on item value is chosen to introduce our approach in the rest of the paper and the problem can be formally formulated as follows:

$$\begin{aligned} Input &: C = \text{knapsack capacity}; N = \text{number of items}; \\ & w_i = \text{weight of item} i; v_i = \text{value of item} i; \\ & M = \text{number of groups}, M \leq N; \\ & G_I = \text{a group of items with the same value}; \\ & V_I = \text{the value of the items in group} I. \\ \text{maximize} & \sum_{i=1}^{N} v_i x_i \\ \text{subject to} & \sum_{i=1}^{N} w_i x_i \leq C \\ & v_i = V_I, \text{if } i \in G_I \\ & x_i \in \{0,1\}, i \in \{1, \cdots, N\}. \end{aligned} \tag{3}$$

5.3 DP reconstruction for MCKP

According to the problem definition given in Eq 3, a new DP solution to solve the MCKP problem is proposed in this work as Eqs. 4 and 5, with the assumption that the items in each group G_i is sorted by their weight in increasing order, and $w_{(i,k)}$ denotes the weight of the k-th item in group G_i.

$$\text{MCKP}[i,j] = \begin{cases} 0, & \text{for } i \text{ or } j \leq 0 \\ \max\limits_{0 \leq k \leq |G_i|} \{\text{MCKP}[i-1, j - \text{sumW}(i,k)] + V_i \cdot k\}, & \text{for } i \neq 0 \end{cases} \tag{4}$$

$$\text{sumW}(i,k) = \begin{cases} \sum_{1 \leq s \leq k} w_{(i,s)}, & \text{for } k > 0 \\ 0, & \text{for } k = 0 \end{cases} \tag{5}$$

Similar to the DP formulation in Eq. 2, $MCKP[i, j]$ represents "the maximum value that can be attained under the knapsack capacity j by choosing the items from groups G_1 up to G_i. Hence, $MCKP[M, C]$ is the solution of an MCKP problem, when the items in all the M groups can be considered without exceeding the knapsack capacity C.

The correctness of Eqs. 4 and 5 can be explained intuitively by the reduction from Eq 2, as shown in Fig. 1. In the figure, the computation of the MCKP table is considered as at row i. Assume the total number of items in $G_1, ..., G_{i-1}$ is n, and the results of the MCKP table at row $i-1$ have been computed. Since MCKP is

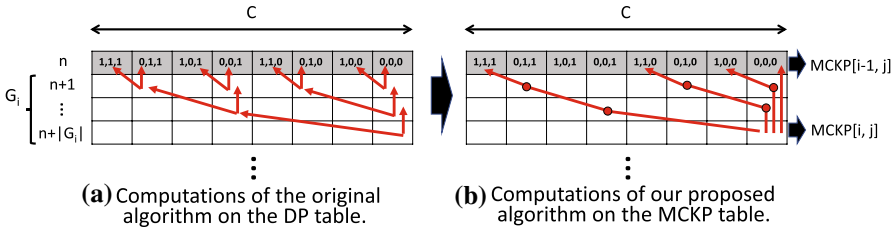


Fig. 1 The reduction process from the original DP algorithm (Eq. 2) to the MCKP DP algorithm (Eqs. 4 and 5) for computing the results of the MCKP table at row i

still a 0/1 knapsack problem, the table in the left of Fig. 1a shows how the i-th row of the MCKP table will be computed according to Eq. 2. Note that because each row in the MCKP table contains a group of items, the computation for the i-th row of the MCKP table maps to multiple rows $(n+1, ..., n+\|G_i\|)$ in the original DP table. The red arrows in the table indicate the computations of Eq. 2 for computing the entry in the bottom right corner. As mentioned above, Eq. 2 intends to enumerate all the possible combinations of choosing the items in group G_i. For instance, G_i has 3 items in Fig. 1, so after three iterations, the results of 8 $(= 2^3)$ possible results are computed as indicated by the tuple of three binary values, $\{b_1, b_2, b_3\}$, shown in the first row of the table where $b_x = 1$ if the x-th item in group G_i is chosen.

In comparison to the result of Eq. 2 in Fig. 1a, the computation of the MCKP algorithm according to Eqs. 4 and 5 is represented in Fig. 1b. Our proposed MCKP algorithm does not enumerate all the combinatorial results from the item selections in group G_i. Instead, only the results from selecting the top k $(k \in \mathbb{Z}^+ + \{0\})$ lightest items in a group are evaluated. This is because any decision that does not select the top k lightest items in a group cannot result in a better solution than our proposed MCKP algorithm as proven in Theorem 2. In order to compute the MCKP table more efficiently, the items in a group are sorted by their weight in increasing order, so the first k items are also the lightest items in a group. Therefore, Eq. 5 computes the total weight of the top k lightest items, and Eq. 4 computes the total value of selecting the top k selections from group G_i under a capacity constraint. Finally, the maximum value from the results of all possible k values determines $MCKP[i, j]$. For instance, Fig. 1b shows that the proposed MCKP algorithm only needs to evaluate and compare four possible results from selecting the top k $(k = 0, 1, 2, 3)$ items from a group, not all the 8 $(= 2^3)$ combinatorial results from 3 group items. Therefore, our approach not only significantly reduces the computation complexity but also enables several performance optimization techniques on GPUs, as detailed in Sect. 5.5.

Theorem 2 *Our proposed MCKP algorithm can obtain the optimal solution because any decision that does not select the top k $(k \in \mathbb{Z}^+ + \{0\})$ lightest items in a group cannot result in a better than the MCKP algorithm.*

Proof Assume S is an optimal solution that does not select the top k ($k \in \mathbb{Z}^+ + \{0\}$) lightest items in a group. That implies S selects an item x from a group G_i, where exist another item $x' \in G_i$ whose weight is smaller than x (i.e., $w_x > w_{x'}$) but not selected in S. Since x and x' are both from the same group, they must have the same value (i.e., $v_x = v_{x'}$). Therefore, another optimal solution S' with the same total value of S is obtained by replacing x with x' in S (i.e., $S' = S \backslash \{x\} \cup \{x'\}$) without exceeding the knapsack capacity limit. Accordingly, for an optimal solution that does not select the top k lightest items in a group, we always find another optimal solution that selects the top k lightest items in the group by repeatedly replacing the selected items with another non-selected item with less weight in the same group. Hence, by evaluating all the solutions that select the top k ($k \in \mathbb{Z}^+ + \{0\}$) lightest items in every group are enough to obtain the optimal solution. □

Theorem 3 *The time complexity of our proposed MCKP algorithm is* $\mathcal{O}(NC + N \log N) \approx \mathcal{O}(NC)$ *since* $C \gg \log N$ *in most of the cases.*

Proof The computation of the DP table is same as the original DP approach of 0/1 knapsack problem which is $\mathcal{O}(NC)$. Sorting items in the same group has the time complexity of $\mathcal{O}(N \log N)$. In result, the time complexity is $\mathcal{O}(NC) + \mathcal{O}(N \log N) = \mathcal{O}(NC + N \log N)$. However, C is larger than $\log N$ in most of the cases, so it can be written as $\mathcal{O}(NC + N \log N) \approx \mathcal{O}(NC)$. □

5.4 GPU implementation

The pseudo-code of our proposed MCKP algorithm on GPUs is shown in Algorithm 2. The CPU code is similar to Boyer's implementation shown in Algorithm 1. Line7-9 is added to classify the items into groups by their values, and Line13 is added to sort the item in a group by their weight in increasing order. Then, a kernel is launched for each group only, instead of each item. Note that all the weights of items in the group, $G[v]$, must be passed to GPU through global memory. In contrast, the GPU code is completely rewritten according to the reconstructed equations in Eqs. 4 and 5. First, in Line24, a chunk of shared memory is allocated in each thread block to load the item weights, W (i.e., $G[v]$ in CPU), from global memory to shared memory. If the number of threads per block (i.e., *blockSize*) is less than the number of items in the group (n), each thread is used to move multiple elements of W to shared memory in Line26-28. Line29–38 implements Eqs. 4 and 5. Specifically, Line 32 implements Eq. 5 to compute *sumW*, and line 36 implements Eq. 4 to keep *maxV*, which is the max of all the top k selection results with k varied from 1 to the group size n. Finally, *maxV* is stored to the entry of the MCKP DP table corresponding to the threadID.

Algorithm 2 The pseudo-code of our proposed MCKP algorithm on GPUs

```
1: CPU process
2: function MAIN_FUNC(N, C, v, w)
3:     Initialize dp0[j] = dp1[j] = 0, for j = 1, ··· , C
4:     Copy_Mem_From_Host_To_Device(dp0)
5:     Copy_Mem_From_Host_To_Device(dp1)
6:     maxV = 0
7:     for i = 1 to N do
8:         G[v_i].push(w_i)
9:         maxV = max{maxV, v_i}
10:    end for
11:    for v = 1 to maxV do
12:        if G[v] ≠ ∅ then
13:            sort_increasing_order(G[v])
14:            Copy_Mem_From_Host_To_Device(G[v])
15:            GPU_kernel_func(dp0, dp1, v, G[v], G[v].size)
16:            swap_mem_pointer(dp0, dp1)
17:        end if
18:    end for
19:    Copy_Mem_From_Device_To_Host(dp0)
20:    Return dp0[C]
21: end function
22: GPU process
23: function GPU_KERNEL_FUNC(dp0, dp1, v, W, n )
24:    __shared__ SMEM[n]
25:    j = threadID
26:    for i = 0 to ⌈n / blockSize⌉ do
27:        SMEM[i · blockSize + j] = W[i · blockSize + j]
28:    end for
29:    sumW = 0
30:    maxV = dp0[j]
31:    for k = 1 to n do
32:        sumW = sumW + SMEM[k]
33:        if j < sumW then
34:            break
35:        end if
36:        maxV = max{maxV, dp0[j − sumW] + v · k}
37:    end for
38:    dp1[j] = maxV
39: end function
```

5.5 Performance optimization

Our proposed MCKP algorithm enables several-performance optimizations on GPU architecture. We discuss each of them as follows:

Reduce the number of kernels and increase the data parallelism Clearly, the number of the launched kernels is reduced from N to M, where N is the number of items, and M is the number of groups. Moreover, by exploring the data parallelism, each kernel can perform more computation workloads without synchronizing with the CPU. With the features of data parallelism and reusability across threads, we introduce an optimization technique by using shared memory.

Utilize shared memory Boyer's method did not utilize the shared memory on GPUs, because there is no reused data within a kernel. On the contrary, after grouping the computation of multiple items in a single kernel, the weight of items is reused among threads to compute $sumW$ in Line32 of Algorithm 2. Hence, the array of item weight, W can be moved to the shared memory for reducing the number of global memory access.

Overlap CPU time with GPU time In order to make the computation on GPUs more efficient, the items in a group must be sorted in our proposed MCKP algorithm. The sorting operation can be done on CPUs or GPUs. The reason this work decided to implement it on CPU is that it can take advantage of the asynchronous call of kernel launch to overlap the sorting time of group G_{i+1} with the kernel execution of group G_i, thus, resulting in reducing the GPU computation time without adding overhead to the overall execution time.

Avoid concurrent writes When parallelizing the computation for computing the value $dp1$ from $dp0$ in the MCKP algorithm, threadID can be used to denote the column index of $dp0$ or $dp1$. When threadID is used as the column index of $dp0$, $dp0[j]$ is used to update $dp1[j + sumW]$ for all possible values of $sumW$ (i.e., $dp1[j + sumW] = \max\{dp0[j + sumW], dp0[j] + v \cdot k\}$). On the other hand, when threadID is used as the column index of $dp1$, $dp1[j]$ is updated by the values of $dp0[j]$ and $dp0[j - sumW]$ (i.e., $dp1[j] = \max\{dp0[j], dp0[j - sumW] + v \cdot k\}$). The first method involves concurrent writes among threads, so it requires the use of atomic operation on max to prevent race conditions. As the second method only involves concurrent reads among threads without race condition, the implementation in Algorithm 2 uses threadID as the column index of $dp1$ not $dp0$ to avoid concurrent writes and the use of atomic operation.

Minimize global memory access Global memory access is the main performance bottleneck for solving the knapsack problem on GPUs. In Boyer's method, the computing results of each iteration (i.e., item) must be written to the DP table on global memory. In contrast, the optimized code only needs to write the computing results of a group of items to the DP table on global memory. The temporal computing results from each item in a group are stored in a register variable $maxV$ instead. Therefore, the CGMA ratio of the GPU implementation of our MCKP algorithm can also be significantly increased from 1 to 5 as proven in Theorem 4.

Prune GPU computations In Line33-34 of Algorithm 2, the loop is broken when $sumW < j$, which means the aggregated weight of the top k selected items already exceeds the current knapsack capacity j. Thus, there is no need to compute the results of selecting even more items. In other words, our proposed MCKP algorithm provides the opportunities to prune the computations and reduce the overall runtime.

Theorem 4 *The CGMA* (*Compute to Global Memory Access*) *ratio of the GPU kernel in our proposed MCKP algorithm shown in Algorithm* 2 *is approximately* 5.

Proof The operations in the kernel code of Algorithm 2 contain two loops. The first loop in Line26–28 is for loading shared memory, and the second loop in Line31–37

is for computing the result. The number of iterations for the first loop is $\lceil \frac{n}{blockSize} \rceil$, while the number of iterations for the second is n. While the value of $blockSize$ is normally in the range of hundreds to thousands, only the CGMA ratio of the second loop is needed to consider.

In each loop iteration (Line23–39), a thread performs 5 operations, including 1 multiplication (i.e., $v \cdot k$ in Line36), 1 max (i.e., in Line36), 1 subtraction (i.e., $j - sumW$ in Line36) and 2 additions (i.e., $sumW + SMEM[k]$ in Line32 and $dp0[] + vk$ in Line36), while only issue 1 global memory access on $dp0[j - sumW]$. Therefore, the CGMA ratio is approximately 5.

Overall, it is easy to see that the performance gain of our MCKP algorithm over Boyer's method grows greater when the size of the group for each item class becomes larger. Also, even in the worst case, when no items share the same weight or value, our MCKP algorithm simply becomes the same as Boyer's method. Therefore, our approach should never perform worse than Boyer's method.

6 Experiment evaluation

6.1 Setup

We conducted the experiments on two different GPU machines. One is a desktop computer equipped with an Intel Core i9-10900 10-Core 4.6GHz CPU and an NVIDIA RTX 3070 GPU card. The other is a GPU node from the TAIWANIA 2 supercomputer [17] equipped with two Xeon Gold 6154 18-Core 3GHz CPUs and 8 Tesla V100 SXM2 GPUs. Their detailed hardware specifications are summarized in Table 1. Note that the computing performance of RTX 3070 GPU (20.3TFLOPS) is higher than the V100 GPU (15.7FLOPS) because the Ampere architecture of RTX 3070 is newer than the Volta architecture of V100. On the contrary, the (global) memory bandwidth of RTX 3070 (448GB/s) is less than half of the bandwidth of V100 (900GB/s). Therefore, comparing the results from the two GPU architectures can further demonstrate the performance impact of MCKP on computing and memory access. The software environment on both machines is kept the same, where the programming tools installed on both machines are CUDA 11.2 and GCC 10.2. For both GPUs, the shared memory size is 49152 Bytes, and the maximize thread block size is 1024. We use the largest thread block size in our experiments because a larger block size can reduce the number of data item that needs to be moved from the global memory to the shared memory per thread, and thus leads to the best execution performance for our algorithm on GPUs. Our GPU code implementation also optimized by using all the basic GPU programming techniques, including unrolling, blocking, memory coarsening, etc. Our experiments only focus on evaluating the performance improvements from our algorithm design.

The benchmarking datasets used in our evaluations are shown in Table 2. In the default setting, the number of items is 10^6, and they are classified into 10^3 groups

Table 1 The hardware specification of GPU machines in our experiments

Machine	Host		GPU (Device)							
	CPU Model	PCIe	Model	Architecture	Cores	FP32 peak op per sec	Memory size (GB)	Memory bandwidth	Price (USD)	Release year
Desktop	Core i9-10900 @4.6 GHz	Gen3x8	RTX3070	Ampere	5888	20.3T @1725MHz	8	448 GB/s	500	2020
TAIWANIA 2	Xeon Gold 6154 @3GHz	Gen3x8	V100 SXM2	Volta	5120	15.7T @1380MHz	32	900 GB/s	>10,000	2017

Table 2 Experimental datasets

Parameter	Description	Default value
N	Number of items	10^6
M	Number of item groups	10^3
w_i	Weight of the i-th item	$rand(1, 10^3)$
v_i	Value of the i-th item	$rand(1, 10^6)$ in M bins
C	Knapsack capacity	$\sum_{n=1}^{N} w_i/10$

by the item values. The weight and value of items are generated independently by a random function with values between 1 and 1,000. To control the number of groups, we randomly mapped items into groups, then randomly assigned the value of groups. The knapsack capacity is determined by the total weight of all items divided by a constant factor 10. For each experiment, the results from 5 randomly generated problem instances were collected and the average numbers reported below.

While Boyer's method is the only known method to implement the DP algorithm for 0/1 knapsack problem, our proposed MCKP method improves its performance through three main optimizations. The first is "Grouping & Pruning", which groups items with the same value to reduce kernels, and prunes the computation within a group when $sumW < j$. Second, "Avoid atomicMax" uses threadID to denote the column index of DP1 instead of DP0 thus avoiding concurrent writes and atomicMax operations. Third, "Shared Memory" loads item weights into shared memory for minimizing global memory access. To show the performance improvement from each of these optimizations, we rewrite the program of Boyer's method and compared to several variants of our method (MCKP) with different optimization configurations. The name and the setting of our compared methods are summarized in Table 3.

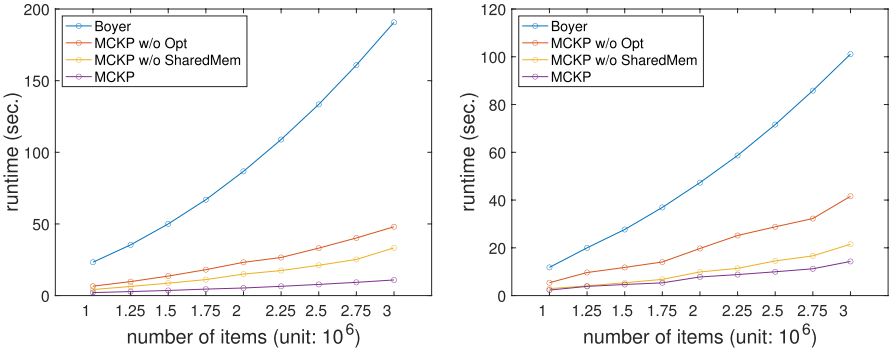
6.2 Performance scalability

The time complexity of 0/1 knapsack problem is known to be $\mathcal{O}(N \cdot C)$, where N is the number of items and C the knapsack capacity. In this set of experiments, we observe the performance under varied problem scales by increasing the number of items (N) from 10^6 to 3×10^6. Because the ratio between the total item weight and

Table 3 The optimization configuration of compared methods

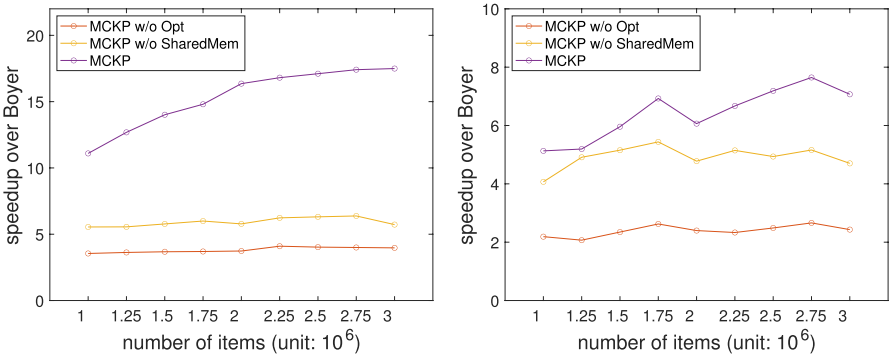
Method	Configurtion	Grouping & pruning	Avoid atomicMax	Shared memory
Boyer [3]	–	No	No	No
MCKP (ours)	w/o Opt	Yes	No	No
	w/o SharedMem	Yes	Yes	No
	–	Yes	Yes	Yes

If the optimization is supported in a method, it is indicated by "Yes"; otherwise, it is indicated by "No"



(a) RTX 3070 **(b)** Tesla V100

Fig. 2 Runtime comparison under $10^6 \sim 3 \times 10^6$ items



(a) RTX 3070 **(b)** Tesla V100

Fig. 3 The runtime speedup over Boyer's method under $10^6 \sim 3 \times 10^6$ items

the knapsack capacity (C) is fixed at 10 in our dataset setting, the knapsack capacity in the experiments is also increased linearly to the number of items. Hence, the problem complexity grows $\mathcal{O}(N^2)$ in these experiments. Note that the number of GPU threads per GPU kernel is the same as the knapsack capacity (C) in all GPU methods, so the parallelism of GPU codes also increases linearly to the number of items.

Figure 2 shows the runtime comparison between Boyer's method and the MCKP algorithms under $10^6 \sim 3 \times 10^6$ items. The corresponding runtime speedup of MCKP over Boyer's method is plotted in Fig. 3. From these plots, we have the following key observations.

First, as expected from the complexity analysis above, the runtime of all methods increases as the problem scale grows with more items. However, as observed in Fig. 2, the increasing rate is over linear, especially under larger scales. We believe this is caused by the growing resource contention from larger problem scales

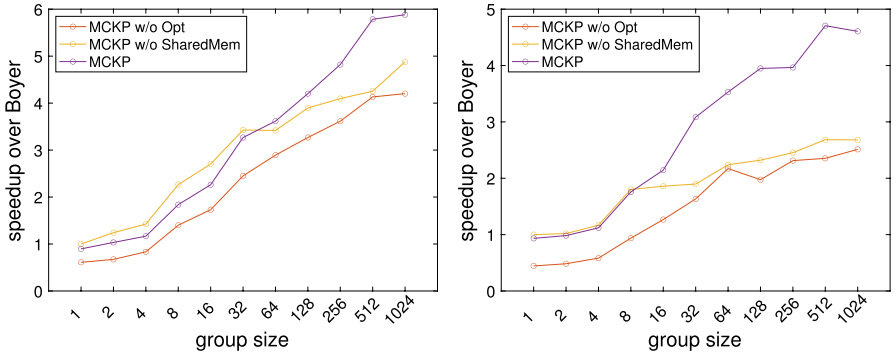
when the total number of threads per kernel increases. As GPU is known to maximize its throughput and efficiency by multiplexing its hardware resources among threads, although more threads are used for a larger problem size, the GPU hardware resources, such as GPU memory bandwidth and cores, are still limited. Hence, higher resource contention and greater performance degradation are expected for the kernels with more threads. As a result, all GPU methods cannot achieve the ideal linear scalability in practice.

Second, regardless of the problem scale, MCKP always achieves higher performance improvement over Boyer's method. Take the result at $N = 10^6$ as an example. *MCKP w/o Opt* achieves 3.5x performance speedup on RTX 3070, and 2.2x performance speedup on Tesla V100. By avoiding the atomicMax operation, *MCKP w/o SharedMem* further increases the speedup on RTX 3070 GPU and Tesla V100 GPU to 5.4x and 4.1x, respectively. After all three optimizations, including shared memory, are applied, the speedup of *MCKP*, can further reach 11.1x and 5.1x on RTX 3070 GPU and Tesla V100 GPU, respectively.

Third, the speedup of MCKP over Boyer's method should only be affected by the average group size of items theoretically. This is because the computations for both MCKP and Boyer's method are expected to grow linearly to the number of items and the size of knapsack capacity. That is why we also observed that the performance speedup of our method without optimization and without shared memory roughly stays constant across all problem scales, as shown in Fig. 3. However, interestingly, we found that the speedup of *MCKP* grows like a concave down function with a growing improvement rate initially but converges to a constant improvement factor at the end. We believe this is the limit of our algorithm since we increase the CGMA ratio from 1 to approximately 5, which is proven in Theorem 4. Our optimization techniques not only reduce the amount of memory access and computations of GPU kernels but also help alleviate the resource contention on global memory and multiprocessor. When the performance speedup is less (e.g., *MCKP w/o Opt*), the difference of resource contention is limited so that no side effect can be observed. However, when the speedup is high enough (e.g., *MCKP*), the data parallelism and reusability help us to gain additional performance improvement. Nevertheless, when the number of items is too large (e.g., over 2.5×10^6), the resource contention overhead may become too large to be affected by the workload reduction from our method. Overall, we observed greater speedup improvement under larger problem scales, and the speedup reaches up to 17.5x and 7.6x on RTX 3070 GPU and Tesla V100 GPU, respectively.

6.3 Grouping analysis

The performance of our approach highly depends on the group size of items because we aim to explore the data reuse pattern and redundant operations among the items in a group. The following is the performance impact of group size on our optimizations.



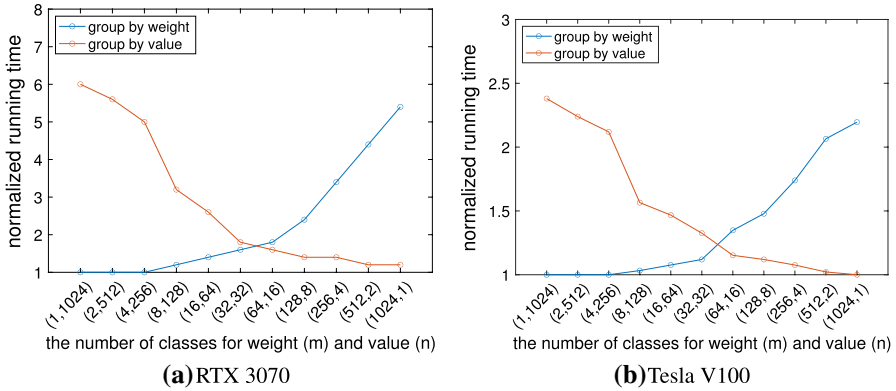
(a) RTX 3070 **(b)** Tesla V100

Fig. 4 The runtime speedup over Boyer's method under varied group sizes from 1 to 1024

Figure 4 shows the runtime speedup of MCKP over Boyer's method on the datasets with varied group sizes. As expected, higher performance improvements are obtained under a larger group size, especially when the shared memory is used. Moreover, we have observed much larger performance improvements on the RTX 3070 than the V100, because the limited bandwidth of global memory presents a greater performance bottleneck on RTX 3070. The speedup on RTX 3070 and V100 can reach up to 5.9x and 4.7x, as the group size increases from 1 to 1024. Interestingly, we also have found the performance of *MCKP w/o SharedMem* could perform better than *MCKP* when the group size is extremely small (≤ 32). This is because the performance benefited from our approach is not enough to overcome the data movement overhead from copying data into shared memory when the group size is too small. However, overall, *MCKP* has comparable or better performance than others at any group size. Furthermore, including the worst-case scenario where the group size is 1, *MCKP* still has the same performance as Boyer's method. Therefore, our approach can consistently outperform Boyer's method.

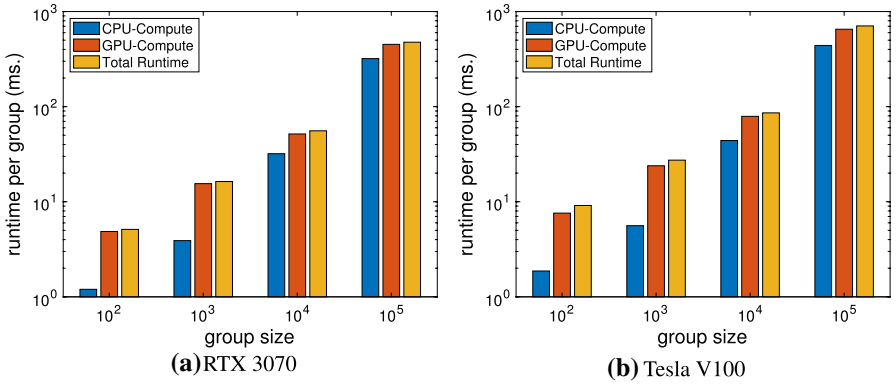
In order to see if MCKP can be applied to the datasets with classes on either item value or item weight, we generated a set of datasets whose item weights and values can be classified into m and n groups, respectively. Note that the total number of items is fixed in this set of experiments, so fewer groups imply a larger group size as well. The runtime of MCKP algorithm with grouping by weight and value on these datasets is plotted in Fig. 5 with the x-axis label (m, n). As expected, the runtime of MCKP algorithm with grouping by value is only affected by the number of groups on item values. By contrast, the runtime with grouping by weight is only affected by the number of groups on item weights. Therefore, the results from *grouping by value* and *grouping by weight* are basically reversed in the plot. Depending on the dataset characteristics, we can apply different grouping strategies to obtain the best performance improvement.

Finally, we evaluate the overhead of CPU sorting from our approach under varied group sizes and prove the CPU sorting time can be overlapped and hidden by the GPU time. Noted, the GPU time includes the data transfer time for moving the



(a) RTX 3070 **(b)** Tesla V100

Fig. 5 The runtime of our MCKP algorithm on a set of datasets whose item weights and values can be classified into m and n groups, respectively. The number of classes for the weights and values of a dataset is varied from 1 to 1024 and labeled by the x-axis (m, n)



(a) RTX 3070 **(b)** Tesla V100

Fig. 6 CPU and GPU compute overlapping

data items from the CPU memory to the GPU global memory. Since our algorithm doesn't change the amount of data movement between CPU and GPU, the data transfer time is only a constant overhead ($< 5\%$) in our experiments. As shown by Algorithm 2, MCKP does introduce additional item sorting time on CPU, the sorting growing $\mathcal{O}(n \log n)$ to the group size n. On the other hand, the GPU time for processing a group of items grows only $\mathcal{O}(n)$. Therefore, as shown by the profiled time plotted in Fig. 6, both CPU time and GPU time grow under a larger group size and the growing rate of CPU time is faster than GPU time. Despite that, the CPU sorting time can be overlapped by the GPU time through CUDA's asynchronous kernel execution. Therefore, the total runtime is almost the same as the GPU time at any group size in the plot, which implies the CPU time can be completely hidden in our approach as long as the CPU time is less than the GPU time. Note that RTX 3070 and V100 are both one of the fastest GPU models, and we didn't parallelize the sorting code on CPUs. Hence, if a slower GPU model is used, the GPU time is likely to

be increased and the CPU sorting time can more easily be overlapped. Therefore, the CPU sorting time won't cause any performance overhead in MCKP.

6.4 GPU architecture comparison

In Sect. 6.2, we discuss the performance impact of individual optimization techniques on different GPU architectures. As shown in Fig. 3, which summarize the performance gain from each optimization technique by the gap of the speedup between compared methods; clearly, the shared memory technique provides a much greater performance improvement than the other two optimization techniques on RTX 3070 as indicated by the huge gap between *MCKP* and *MCKP w/o SharedMem*. This is because the shared memory technique can reduce the number of global memory access by five-fold, as proven in Theorem 4. As known from the hardware specification in Table 1, RTX 3070 is a mid-end gaming GPU card with the price of $500, so it is a memory-bound GPU with much less memory bandwidth than computing capability. Therefore, the shared memory technique can mitigate the memory bottleneck problem and deliver much higher computing throughput. While V100 is targeted for data center, it has a more balanced computing and memory access performance, so the shared memory technique has a lower performance gain on V100.

Furthermore, we compared the runtime of the exact GPU implementation on RTX 3070 and V100 as indicated in Fig. 7. If the speedup of a method is larger than 1 in the plot, it means the method is running faster on RTX 3070 than V100. As expected, the speedup between the two GPU cards for any method is irrelevant to the problem scale because it should only depend on the computing capability of the GPU cards. We also found that V100 has better performance than RTX 3070 for all the methods, except *MCKP*. It is known that RTX 3070 has better computing performance but worse memory bandwidth than V100. Since knapsack DP algorithms are memory-bound applications for GPUs, it is no surprise to see V100 has better performance. The speedup of Boyer's method running on RTX 3070 over V100 is nearly equal to 0.5, which matches the ratio of their global memory bandwidth ($\frac{900}{448} \simeq 0.5$), while the speedup of the MCKP has the average of 1.3, which

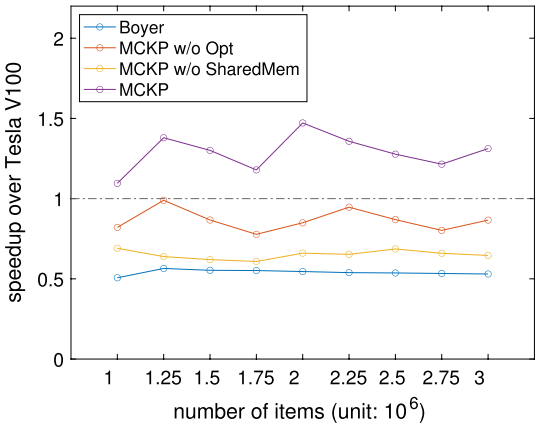


Fig. 7 The speed ratio comparison between RTX 3070 and Tesla V100. The ratio is defined as the runtime on V100 divided by the runtime on RTX 3070

is the ratio of their computing throughput ($\frac{20.3T}{15.7T} \simeq 1.3$). By observing the runtime ratio of two GPUs, the MCKP method achieved higher performance on RTX 3070 than V100 what proves our optimizations to be able to solve the memory bottleneck problem and allow GPUs with higher computing throughput to deliver better application performance.

We also analyzed the performance statistics reported by the NVIDIA CUDA profilers, *nvprof* on V100 and *Nsight Compute* on RTX 3070, using the default dataset specified in Table 2. The *instructions per cycle (ipc)* to present the GPU computing performance and the *load and store throughput* on global memory to present the memory performance were selected. As shown in Table 4 which reports their performance measurements, we have three improvements. First, the MCKP method significantly improves the ipc on both GPU models, which implies the memory access overhead is minimized, and computing throughput is maximized. Second, the load throughput is increased because the kernel can compute multiple iterations at once; this indicates the data parallelism is increased, in contrast with Boyer's method which requires a kernel synchronization between GPU and CPU after each iteration that throttles the throughput for both computing and memory access. Last, the store throughput becomes lower due to fewer write operations. By writing the temporary computing results from each iteration to a register, *maxV*, instead of global memory everytime, global memory writes only occur at the end of a kernel execution after the computations of multiple iterations. Overall, the profiling results proved that our approach can maximize GPU throughput and minimize memory bottleneck by reducing the memory access operations and improving the throughput of global memory.

7 Conclusion

This work presents an optimized DP algorithm and implementation to maximize the performance for solving *Multi-Class 0/1 Knapsack Problem* (MCKP) on GPUs. The main contribution of MCKP is to explore data parallelism and the reusability across threads. Both the computing complexity and runtime performance are compared between our proposed approach and the well-known Boyer's method. We proved our method could improve the CGMA (Compute to Global Memory Access) ratio 5-fold, increasing from 1 to 5. The runtime performance comparison was extensively evaluated on two modern GPU models, RTX 3070 and V100. Comparing to Boyer's method, our approach achieved up to 18x speedup on RTX 3070, and 8x speedup

Table 4 The comparison of performance metrics reported by the NVIDIA CUDA profilers

GPU	Method	ipc	Load throughput	Store throughput
RTX 3070	Boyer	0.15	409 GB/s	186.54 GB/s
	MCKP	0.73	3100 GB/s	1.05 GB/s
V100	Boyer	0.94	589 GB/s	279.60 GB/s
	MCKP	2.60	2977 GB/s	0.85 GB/s

on V100, and greater speedup improvements can be observed under larger problem scales with an increasing number of items and knapsack capacity. According to the hardware specification, RTX 3070 has a higher computing throughput but lower memory bandwidth than V100. Furthermore, the improvement of our approach is also much higher on RTX 3070 than V100 because the memory performance bottleneck problem is worse on RTX 3070 than V100. Therefore, our optimized implementation successfully achieves better performance on RTX 3070 than V100 after having overcome the memory bottleneck. In contrast, Boyer's method suffers from the memory access overhead and cannot obtain better performance on RTX 3070 than V100. The performance metrics reported by NVIDIA CUDA profilers also confirm our approach can significantly increase the instructions per cycle (ipc) on both GPU models and improve the memory load throughput while minimizing memory write operations. The result of our work demonstrates the importance of addressing the memory access overhead on GPUs and provides a more efficient GPU solution for solving the 0/1 knapsack problem in a wide range of application domains.

Acknowledgements We thank to National Center for High-performance Computing (NCHC) for providing computational and storage resources. We also thank to Prof. Ing-Jer Huang from National Sun Yat-sen University for providing valuable insights and comments to our work.

References

1. Bellman R (1966) Dynamic programming. Science 153(3731):34–37. https://doi.org/10.1126/science.153.3731.34
2. Boukedjar A, Lalami ME, El-Baz D (2012) Parallel branch and bound on a cpu-gpu system. In: 2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing, pp. 392–398. https://doi.org/10.1109/PDP.2012.23
3. Boyer V, El Baz D, Elkihel M (2012) Solving knapsack problems on gpu. Comput Op Res 39(1):42–47
4. Carneiro T, Muritiba AE, Negreiros M, Lima de Campos GA (2011) A new parallel schema for branch-and-bound algorithms using gpgpu. In: 2011 23rd International Symposium on Computer Architecture and High Performance Computing, pp. 41–47. https://doi.org/10.1109/SBAC-PAD.2011.20
5. Ding N, Williams S (2019) An instruction roofline model for gpus. In: 2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), pp. 7–18. https://doi.org/10.1109/PMBS49563.2019.00007
6. Garey MR, Johnson DS (1990) Computers and intractability; a guide to the theory of NP-completeness. W. H Freeman & Co., New York
7. Hajarian M, Shahbahrami A, Hoseini F (2016) A parallel solution for the 0-1 knapsack problem using firefly algorithm. In: 1st Conference on Swarm Intelligence and Evolutionary Computation (CSIEC), pp. 25–30. https://doi.org/10.1109/CSIEC.2016.7482134
8. HPC Advisory Council: The Top 500 List (2021). https://www.top500.org/lists/top500/2021/06/
9. Huang S, Xiao S, Feng W (2009) On the energy efficiency of graphics processing units for scientific computing. In: 2009 IEEE International Symposium on Parallel Distributed Processing, pp. 1–8. https://doi.org/10.1109/IPDPS.2009.5160980
10. Kelly T (2005) Generalized knapsack solvers for multi-unit combinatorial auctions: Analysis and application to computational resource allocation. In: P. Faratin, J.A. Rodríguez-Aguilar (eds.) Agent-Mediated Electronic Commerce VI. Theories for and Engineering of Distributed Mechanisms and Systems, pp. 73–86. Springer Berlin Heidelberg, Berlin, Heidelberg

11. Konstantinidis E, Cotronis Y (2015) A practical performance model for compute and memory bound gpu kernels. In: 2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, pp. 651–658. https://doi.org/10.1109/PDP.2015.51
12. Kumaraguruparan N, Sivaramakrishnan H, Sapatnekar SS (2012) Residential task scheduling under dynamic pricing using the multiple knapsack method. In: 2012 IEEE PES Innovative Smart Grid Technologies (ISGT), pp. 1–6. https://doi.org/10.1109/ISGT.2012.6175656
13. Lalami ME, El-Baz D (2012) Gpu implementation of the branch and bound method for knapsack problems. In: IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum, pp. 1769–1777. https://doi.org/10.1109/IPDPSW.2012.219
14. Lee J, Shragowitz E, Sahni S (1988) A hypercube algorithm for the 0/1 knapsack problem. J Parallel Distrib Comput 5(4):438–456. https://doi.org/10.1016/0743-7315(88)90007-X
15. Lin J, Storer JA (1991) Processor-efficient hypercube algorithms for the knapsack problem. J Parallel Distrib Comput 13(3):332–337. https://doi.org/10.1016/0743-7315(91)90080-S
16. Liu H, Shao Z, Wang M, Du J, Xue CJ, Jia Z (2009) Combining coarse-grained software pipelining with dvs for scheduling real-time periodic dependent tasks on multi-core embedded systems. J Signal Process Syst 57(2):249–262. https://doi.org/10.1007/s11265-008-0315-2
17. National Center for High-performance Computing: TAIWANIA2 (2018). https://www.nchc.org.tw/
18. Nawaz Z, Stefanov T, Bertels K (2009) Efficient hardware generation for dynamic programming problems. In: 2009 International Conference on Field-Programmable Technology, pp. 348–352. https://doi.org/10.1109/FPT.2009.5377618
19. NVIDIA: NVIDIA A100 datasheet (2020). https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet.pdf
20. NVIDIA: Cuda c++ programming guide (2021). https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf
21. Oak Ridge National Laboratory: SUMMIT (2018). https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/
22. O'Connell JF, Mumford CL (2014) An exact dynamic programming based method to solve optimisation problems using gpus. In: Second International Symposium on Computing and Networking, pp. 347–353. https://doi.org/10.1109/CANDAR.2014.27
23. Odlyzko AM (1990) The rise and fall of knapsack cryptosystems. In: In Cryptology and Computational Number Theory, pp. 75–88. A.M.S
24. O'Leary DE (1995) Financial planning with 0–1 knapsack problems, part i: domination results. Adv Math Program Financ Plan 4:139–150
25. Pospichal P, Schwarz J, Jaros J (2010) Parallel genetic algorithm solving 0/1 knapsack problem running on the gpu. In: Proceedings of the 16th International Conference on Soft Computing (MENDEL), pp. 64–70
26. Schryen G (2020) Parallel computational optimization in operations research: a new integrative framework, literature review and research directions. Eur J Oper Res 287(1):1–18. https://doi.org/10.1016/j.ejor.2019.11.033
27. Shen J, Shigeoka K, Ino F, Hagihara K (2017) An out-of-core branch and bound method for solving the 0-1 knapsack problem on a gpu. In: International Conference on Algorithms and Architectures for Parallel Processing, pp. 254–267. https://doi.org/10.1007/978-3-319-65482-9_17
28. Shen J, Shigeoka K, Ino F, Hagihara K (2019) Gpu-based branch-and-bound method to solve large 0–1 knapsack problems with data-centric strategies. Concurr Comput Pract Exp 31(4):e4954
29. Sun X, Wu CC, Chen LR, Lin JY (2018) Using inter-block synchronization to improve the knapsack problem on gpus. Int J Grid High Perform Comput (IJGHPC) 10(4):83–98
30. Suri B, Bordoloi UD, Eles P (2012) A scalable gpu-based approach to accelerate the multiple-choice knapsack problem. In: Design, Automation Test in Europe Conference Exhibition (DATE), pp. 1126–1129. https://doi.org/10.1109/DATE.2012.6176665
31. Thant Sin ST (2021) The parallel processing approach to the dynamic programming algorithm of knapsack problem. In: 2021 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (ElConRus), pp. 2252–2256. https://doi.org/10.1109/ElConRus51938.2021.9396489
32. Toth P (1980) Dynamic programming algorithms for the zero-one knapsack problem. Computing 25:29–45

33. Ulm DR, Baker JW (1996) Solving a 2d knapsack problem on an associative computer augmented with a linear network. In: in Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications, pp. 29–32
34. Wang Q, Chu X (2020) Gpgpu performance estimation with core and memory frequency scaling. IEEE Trans Parallel Distrib Syst 31(12):2865–2881. https://doi.org/10.1109/TPDS.2020.3004623
35. Wen H, Zhang W (2015) Exploring shared memory and cache to improve gpu performance and energy efficiency. In: Sixteenth International Symposium on Quality Electronic Design, pp. 402–405. https://doi.org/10.1109/ISQED.2015.7085459
36. Williams S, Waterman A, Patterson D (2009) Roofline: an insightful visual performance model for multicore architectures. Commun ACM 52(4):65–76. https://doi.org/10.1145/1498765.1498785
37. Xiao S, Feng Wc (2010) Inter-block gpu communication via fast barrier synchronization. In: 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS), pp. 1–12. https://doi.org/10.1109/IPDPS.2010.5470477
38. You Y, Zhang Z, Hsieh CJ, Demmel J, Keutzer K (2018) Imagenet training in minutes. In: Proceedings of the 47th International Conference on Parallel Processing, ICPP 2018, pp. 1–10. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3225058.3225069