An Out-of-Core Branch and Bound Method for Solving the 0-1 Knapsack Problem on a GPU

Jingcheng Shen[1], Kentaro Shigeoka[2], Fumihiko Ino[1], and Kenichi Hagihara[1]

[1] Graduate School of Information Science and Technology, Osaka University
1-5 Yamadaoka, Suita, Osaka 565-0871, Japan
jc-shen@ist.osaka-u.ac.jp
[2] Hitachi High-Technologies Corporation
14-24-1 Nishishinjuku, Minato-ku, Tokyo 105-8717, Japan

Abstract. In this paper, we propose an out-of-core branch and bound (B&B) method for solving the 0-1 knapsack problem on a graphics processing unit (GPU). Given a large problem that produces many subproblems, the proposed method dynamically swaps subproblems out to CPU memory. We adopt two strategies to realize this swapping-out procedure with minimum amount of CPU-GPU data transfer. The first strategy is a GPU-based stream compaction strategy that reduces the sparseness of arrays. The second strategy is a double buffering strategy that hides the data transfer overhead by overlapping data transfer with GPU-based B&B operations. Experimental results show that the proposed method can store 33.7 times more subproblems than the previous method, solving twice more instances on the GPU. As for the stream compaction strategy, an input-output separated scheme runs 13.1% faster than an input-output unified scheme.

Keywords: Out-of-core computation, branch and bound, knapsack, GPU

1 Introduction

The 0-1 knapsack problem [1], a combinatorial optimization problem, appears in a wide range of fields such as manufacturing, logistics, and finance. Given n (≥ 1) items, each with its profit and weight, the problem is to determine which item should be included in a knapsack such that the total weight is not beyond capacity c and the total profit is as large as possible:

$$\begin{aligned} &\text{maximize} && \sum_{i=1}^{n} p_i x_i \\ &\text{subject to} && \sum_{i=1}^{n} w_i x_i \leq c,\ i \in \{1, 2, \ldots, n\}, \end{aligned}$$

where p_i and w_i are the profit and weight of the i-th item, respectively, and $x_i \in \{0, 1\}$ is the binary decision variable that decides if the i-th item will be included or not: $x_i = 1$ if included and $x_i = 0$ otherwise. The B&B [2] approach, a

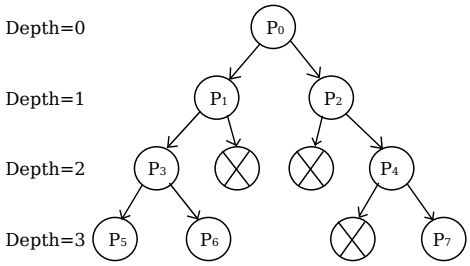

Fig. 1. A search tree of B&B approach. In this figure, node P_i refers to the i-th subproblem. Especially, the root P_0 refers to the original problem. Pruned subproblems (i.e., passive subproblems) are marked with "X."

widely-known scheme for obtaining optimal solutions for NP-hard combinatorial optimization problems, is the most common way to effectively find the optimal solutions of knapsack problems [1]. This iterative scheme reduces the search space by using upper and lower bounds updated every time iterations continue. Each iteration consists of branching, bounding, and pruning operations. The branching operation decomposes a problem into multiple small subproblems. The bounding operation then computes the lower and upper bounds for each subproblem. Finally, the pruning operation discards unpromising (passive) subproblems whose upper bounds are smaller than the best solution found so far. The remained promising (active) subproblems proceed to the next iteration to further apply these three operations. The outline of B&B approach is shown in Fig. 1.

Many B&B approaches successfully solved knapsack problems by exploiting data parallelism using various parallel machines such as single-instruction multiple-data (SIMD) machines [3], cluster systems [4], computational grids [5, 6], and graphics processing units (GPUs) [7–10]. Among these parallel machines, the GPU [11] is a powerful accelerator device for not only graphics applications but also compute- and memory-intensive applications [12, 13]. However, the capacity of GPU memory, 12 GB for example, is relatively small as compared to that of CPU memory. Furthermore, previous GPU-based methods use the GPU memory to manage subproblems, so that these methods fail to solve large problems that rapidly consume GPU memory due to splitting subproblems.

Therefore, in this paper, we present an out-of-core B&B method for solving large 0-1 knapsack problems that exhaust the GPU memory due to many subproblems to be investigated. Our out-of-core method relaxes the limitation of problem size (i.e., maximum number of subproblems) from the capacity of GPU memory to that of CPU memory. To solve a large problem, our method buffers the data of subproblems in the CPU memory rather than the GPU memory. However, such a CPU-centric subproblem management scheme increases the CPU-GPU data transfer that impairs the performance of GPU acceleration. For the purpose of alleviating this side effect, we integrate two optimization strategies into our method: (1) a stream compaction strategy accelerated on the GPU

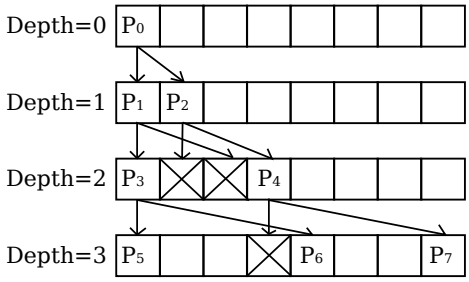

Fig. 2. An array-based subproblem management scheme. Pruned subproblems (i.e., passive subproblems) marked with "X" make the array sparse as the depth of the search tree increases.

completely and (2) a double buffering strategy required for efficient pipelining. Stream compaction here is an important process that converts sparse arrays into dense arrays for reducing the data to be transferred as well as achieving full utilization of massive GPU cores. In order to find an efficient scheme for the stream compaction strategy, we compare a separated scheme with a unified scheme. The separated scheme differentiates an output array from the input array to store the compacted data. On the contrary, the unified scheme uses a single array for both input and output. As such, the unified scheme allows the GPU to simultaneously process 1.5-2 times more subproblems than the separated scheme. As for the double buffering strategy, it hides the data transfer overhead by overlapping data transfer with GPU-based B&B operations.

Following this introduction, Section 2 presents related studies of this paper. Section 3 then summarizes GPU-based B&B methods. Section 4 describes our method and Section 5 shows experimental results. Finally, Section 6 comprises conclusion and future work.

2 Related Work

Boukedjar *et al.* [8] presented a B&B approach that solves the 0-1 knapsack problem on a GPU. Their method managed subproblems in the GPU memory with arrays standing for subproblems. Their method describes a subproblem with several attributes such as the upper and the lower bounds of profit, and currently obtained profit and weight. They prepared an array for each of those attributes. However, those arrays may get sparse after B&B operations (Fig. 2), because elements representing passive subproblems will never be referred again. The sparseness wastes the GPU memory with useless data and harms the locality of reference. Therefore, they integrated a CPU-based compaction strategy into their method for reducing the sparseness. The basic idea for their compaction strategy is to exchange passive elements and active elements with a quicksort-wise scheme, so that all the active subproblems are stored continuously in the front of the array. Because this is a CPU-based sequential operation, their

method transfers all attributes from the GPU memory to the CPU memory every time the compaction proceeds. Therefore, CPU-GPU data transfer remains as a performance bottleneck.

To reduce the amount of data transfer between the CPU and GPU, Lalami *et al.* [9] presented an extension of [8] by restricting transferred data to a single attribute: label data (i.e., label array), where a label shows whether an element is active ($label = 1$) or passive ($label = 0$). They focused on the data access pattern required for compaction. That is, any attribute array generates the same pattern as those generated by other attribute arrays. As such, they computed the pattern on the CPU, according to the label data, and reused that information to carry out stream compaction for every attribute array on the GPU. This extension successfully doubled the performance compared to the baseline method [8]. However, it is still inevitable to transfer data between the GPU and the CPU every time the stream compaction proceeds. Moreover, their method can solve larger problems if subproblems are managed in the CPU memory.

Carneiro *et al.* [14] presented a GPU-accelerated B&B approach that uses a hybrid scheme of breadth first search and depth first search to solve the symmetric traveling salesman problem. Their method initially performs breadth first search on the CPU to generate many initial subproblems to be examined in parallel. After that, the method switches to depth first search that processes B&B operations on the GPU. However, similar to [8], the stream compaction strategy was not integrated into this approach.

3 B&B Approach for Solving Knapsack Problem

The branching operation of the B&B approach is to divide one n-variable (item) problem into two $(n-1)$-variable subproblems. These two subproblems respectively stand for the two cases of decision for the i-th item, where $1 \leq i \leq n$. For an n-item knapsack problem, from the first item to the last item, we iteratively divide one problem into two subproblems with a breadth-first search scheme.

On the other hand, the bounding operation is to reduce the search space by judging each subproblem whether it is possible to get an optimal solution or not. For this purpose, the bounding operation computes the upper and the lower bounds for every existing subproblem. A subproblem whose upper bound is smaller than the best (i.e., biggest) lower bound is passive and must be deleted from the search space. In the following discussion, we assume that items are sorted according to decreasing profit per ratio. This assumption facilitates lower bound computation mentioned below. Let k be the index of the current item to be examined for decision. Let also I_v be the set of items picked for a subproblem, i.e., a vertex v in the search tree. The weight and profit of vertex v, W_v and P_v, respectively, then can be computed as follows:

$$W_v = \sum_{i \in I_v} w_i, \tag{1}$$

$$P_v = \sum_{i \in I_v} p_i. \tag{2}$$

A vertex v can be described as a tuple $(W_v, P_v, U_v, L_v, S_v)$, where U_v and L_v represent an upper and lower bounds of the vertex, respectively, and S_v is the slack variable [9] for the vertex such that

$$\sum_{i=k+1}^{S_v-1} w_i \leq c - w_v < \sum_{i=k+1}^{S_v} w_i. \tag{3}$$

That is, the slack variable S_v is determined by picking all items after the k-th item as long as the knapsack can. Using the slack variable, the residual capacity r of the knapsack is given by:

$$r = c - W_v - \sum_{i=k+1}^{S_v-1} w_i. \tag{4}$$

A lower bound L_v can be obtained by picking the abovementioned items (i.e., from $k+1$ to $S_v - 1$) and others in a greedy manner:

$$L_v = P_v + \sum_{i=k+1}^{S_v-1} p_i + \sum_{i=k+1}^{n} p_i x_i, \tag{5}$$

where $x_i = 1$ if $w_i \leq r - \sum_{j=S_v+1}^{i-1} w_j x_j$. With respect to an upper bound U_v, on the other hand, we adopt the Dantzig bound [15] that can be given by:

$$U_v = P_v + \sum_{i=k+1}^{S_v-1} p_i + \lfloor r P_{S_v} / W_{S_v} \rfloor. \tag{6}$$

4 Proposed Method

The proposed method deploys a CPU-centric subproblem management scheme that exploits the large capacity of the CPU memory to relax the limitation of problem size. However, this out-of-core method can increase the amount of CPU-GPU data transfer, so that we integrate two strategies into our method to mitigate this side effect. The first strategy is a stream compaction strategy that proceeds on GPU completely. As such, it eliminates the data transfer for the pattern computation and increases the parallelism of the pattern computation, compared to [9]. The second strategy is a double buffering strategy that pipelines a series of operations. The strategy hides the overhead of CPU-GPU data transfer by overlapping the GPU-based B&B operations with the CPU-GPU data transfer.

4.1 CPU-centric Subproblem Management

For an efficient CPU-centric subproblem management, the data structure that stores subproblems must satisfy two requirements.

(1) The data structure must be accessed randomly by a GPU thread with $\mathcal{O}(1)$ time.
(2) The data structure stores subproblems continuously without fragmentation in order to improve the efficiency of direct memory access (DMA) required for CPU-GPU data transfer.

To satisfy requirement (1), we choose arrays as a buffer for storing subproblems in the CPU memory. To satisfy requirement (2), we organize the buffer in a circular manner with two pointers: pointer *head* pointed to the first subproblem in the buffer and pointer *tail* pointed to the position behind the last subproblem in the buffer. For every iteration, we transfer a series of subproblems that begin with the subproblems pointed by head from the CPU memory to the GPU memory in order to process GPU-based B&B operations. On the other hand, when we predict that the GPU memory will be exhausted after the next branching operation, we transfer the subproblems from the GPU memory to the CPU memory, stored from the position pointed by tail. Every time a data transfer operation proceeds, the position of head or tail (according to the operation) is updated by an increment of the amount of transferred data. Furthermore, the circular manner means that if we reach the last position of the buffer during transferring data, we turn to its first position again. However, in this occasion, we must call the data transferring function once more, because of the change of the baseline address.

These transfer and computation jobs proceed iteratively until all the subproblems are finished, meaning that decision of the last item, i.e., x_n, is done for every subproblem. Moreover, the proposed method either prunes a finished subproblem if it is judged as passive, or set aside the finished subproblem if it is judged as active currently. In no case does the proposed method branch a finished subproblem.

4.2 Stream Compaction Strategy

To avoid extra CPU-GPU data transfer when computing the data access pattern required for compaction, we adopt a stream compaction strategy that proceeds on the GPU completely. Our stream compaction strategy computes the data access pattern on the GPU instead of the CPU. Moreover, we compare two stream compaction variations, a separated scheme and a unified scheme. The separated scheme requires that the input array (i.e., the array to be compacted) is separated from the output array. As such, the separated scheme consumes more GPU memory than the unified scheme. However, the separated scheme is cogent and it avoids any preprocess. On the other hand, the unified scheme compacts the input array within itself, instead of using a separated output array. Therefore, given that we adopt a double buffering strategy, the buffer size of the unified scheme is 1.5 times bigger than that of the separated scheme. However, the unified scheme requires a preprocess phase for compaction, which incurs an extra overhead. The details of both schemes are presented below.

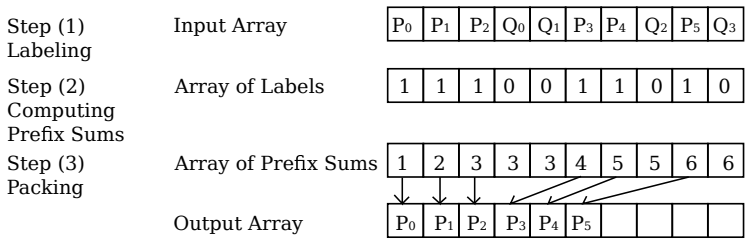

Fig. 3. An overview of the separated stream compaction scheme. In this figure, P_i refers to an active subproblem while Q_i refers to a passive subproblem. In step (1), we associate active and passive subproblems with 1 and 0, respectively. In step (2), we then apply the computation of prefix sums to the array of labels, in order to obtain the array of prefix sums that indicates where to move the active subproblems (i.e., the data access pattern). Finally, in step (3), we contiguously pack the active subproblems into the output array according to the array of prefix sums.

Separated Scheme. The separated stream compaction is a filtering process. The elements that meet the requirements will be selected from the input array, and then be packed into the output array. Generally, the compaction process is done in three steps (Fig. 3): (1) preparation of the array of labels, (2) computation of prefix sums, and (3) packing procedure. We use the thrust library [16] to implement prefix sums computation.

Unified Scheme. The unified scheme overwrites the input array by immediately moving some active elements onto passive elements (Fig. 4). The general idea is to calculate the number of active elements n_a that is used as a border index. We can predict that all active elements will be stored in the first n_a elements of the input array after compaction. Thus, we only need to move the active elements stored after the border index onto the passive elements stored before the border index. The unified scheme requires preparation of the array of labels, computation of prefix sums, and packing procedure, the same as the separated scheme. Besides, the unified requires a preprocess phase before the packing procedure. This preprocess phase consists of two steps:

(1) The unified scheme computes the number of active elements (n_a) that is used as the border index.
(2) The unified scheme associates active elements with passive elements by calculating the order number of every active element stored after the border index and the order number of every passive element stored before the border index. For instance, the i-th active element stored after the border index is associated with the i-th passive element stored before the border index. By doing so, the unified scheme prepares a lookup table for looking up the destinations where the active elements should be moved.

Algorithm 1 describes the lookup table preparation procedure.

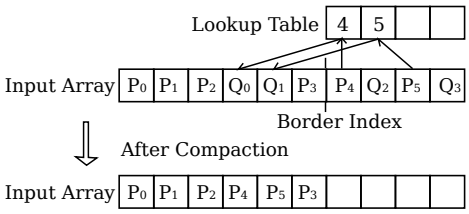

Fig. 4. An overview of the unified stream compaction scheme based on a lookup table.

Algorithm 1: Preparation for a lookup table.

Input: (1)*Labels*, the array of labels and (2) *Sums*, the array of prefix sums
Output: *Table*, the lookup table

```
1 i := threadID
2 if n_a ≤ i then
3     return
4 end if
5 l := Labels[i]
6 if l = 0 then
7     passive_order_number := i − Sums[i]
8     Table[passive_order_number] := i
9 end if
```

4.3 Double Buffering Strategy

In order to hide the overhead of CPU-GPU data transfer, we realize a double buffering strategy that overlaps the CPU-GPU data transfer with GPU-based B&B operations using two CUDA streams (Fig. 5). A CUDA stream here is a sequence of operations that execute in issue-order on the GPU [17]. The data transfer proceeds on a stream using buffer A while the GPU-based B&B operations proceed on the other stream using buffer B. When operations on both streams complete, the strategy exchanges the references of buffer A and buffer B (i.e., pointer A and pointer B in Fig. 5), and then proceeds to the next iteration of this overlapped transfer and computation procedure until finishing processing all subproblems left in the circular buffer. However, before the next iteration, the strategy must synchronize both streams to ensure operations of the current iteration have finished. We carry out synchronization to prevent both streams from simultaneous data transfer, lest the circular buffer is inconsistent.

Besides, we appropriately use two execution modes according to the number of subproblems managed on the CPU. If the circular buffer lacks sufficient subproblems to fully exploit the massive parallelism on the GPU, we only use CPU to process the branching, bounding, and pruning operations. We use GPU to process the operations otherwise. We experimentally determined $24,576$ as the threshold number of subproblems that triggers changing the execution mode. The threshold number may vary under different environments.

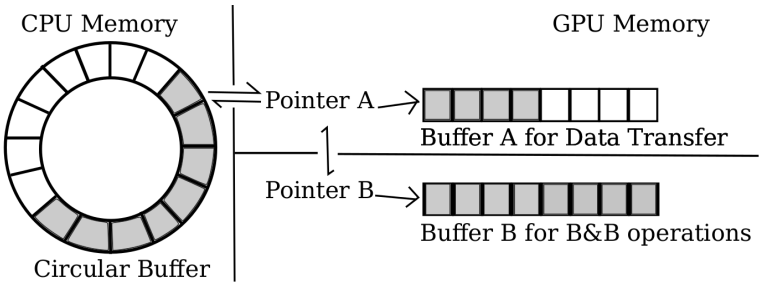

Fig. 5. An overview of the double buffering strategy, which overlaps the CPU-GPU data transfer with GPU-based B&B operations using two CUDA streams.

5 Experimental Results

We conducted experiments to evaluate the proposed method in terms of the problem size and the execution time. We used Lalami's method [9] as a comparable method. Both the proposed method and the previous method are honestly implemented with the same optimization techniques. As for the stream compaction strategy, we compared the separated scheme with the unified scheme in terms of the performance. For datasets, we used a suite of benchmarking datasets [18, 19] shown in Table 1. We used strongly correlated instances with up to $n = 1000$ items, where the weight of each item mainly determines its profit. Such instances can be efficiently parallelized because of enormous combinations of items that may be an optimal solution, and therefore, represent the most difficult problems. We generated 20 different instances for each n.

Table 2 shows the specifications of two experimental machines. We prepared two machines equipped with different capacities of GPU memory: 2 GB and 12 GB for machines 1 and 2, respectively. These capacities were relatively small for the datasets, which consumed at most 20 GB of memory space. Both machines ran on the Ubuntu 14.04 with CUDA 7.5 [20]. Besides, the version of thrust library [16] was 1.8.3.

Table 1. Experimental datasets.

Parameter	Value
p_i: profit of the i-th item	$w_i + 1000 + \text{random}(-20, 20)$
w_i: weight of the i-th item	$\text{random}(1, 10000)$
c: knapsack capacity	$\sum_{i=1}^{n} w_i * 100/1001$

Table 2. Specification of experimental machines.

	Machine 1	Machine 2
CPU	Intel Core i7-6700	Intel Xeon E5-2660 v3
CPU memory capacity	32 GB	64 GB
GPU	GeForce GTX 680 Kepler	GeForce GTX Titan X Pascal
GPU memory capacity	2 GB	12 GB
GPU driver version	375.39	352.39
PCIe Controller	gen3 $\times 16$	gen3 $\times 16$

5.1 Robustness Against the Increase of Problem Size

We prepared ten classes of instances, varying from 100 items to 1000 items. We generated 20 different instances for each class. In this experiment, we allocated an array of 20 GB for the circular buffer. On the other hand, two arrays of total 2 GB were used for processing B&B and stream compaction operations on the GPU. As shown in Fig. 6, our CPU memory subproblem management scheme successfully solved more instances, demonstrating more robustness against the increase of problem size. This robustness is noticeable especially for the "properly" large instances, varying from 400 items to 700 items. However, there was no significant difference for the instances of more than 700 items because sharply increasing subproblems beat the capacity of CPU memory, resulting in a memory exhaustion. Moreover, we can conclude that our out-of-core method solved 33.7 times more subproblems than the previous in-core method. For example, our method successfully stored $845,940,791$ subproblems at a time, whereas the number of subproblems was limited by $25,096,462$ in the previous method.

5.2 Performance Comparison

For each instance class, from the instances which can be solved by both methods, we selected a representative one to compare the execution time of the two methods. For example, presuming that in the 700-item instance class there were three instances which can be solved by both methods, we first computed the average execution time of the three instances processed by the previous method. We then selected the instance whose execution time was nearest to the average execution time (Fig. 7). Obviously, the instances that can be solved by both methods did not produce so many subproblems that exhaust the GPU memory, thus, there is no extra data transfer for swapping-out operations. Our method achieved a higher performance because our stream compaction strategy (with the separated scheme) was completely processed on the GPU in parallel. As presented in Sect. 2, the previous stream compaction strategy computed data access pattern on the CPU, sequentially instead of in parallel, and it required to transfer the array of label between the CPU and GPU. Therefore, our stream compaction strategy ran 4.5 times faster than the previous strategy, shortening the total execution time by 26% on average.

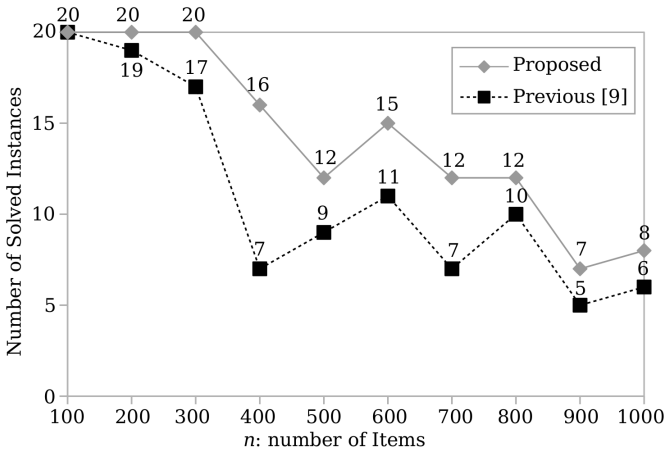

Fig. 6. Comparison of the numbers of instances solved by the proposed method and the previous method [9].

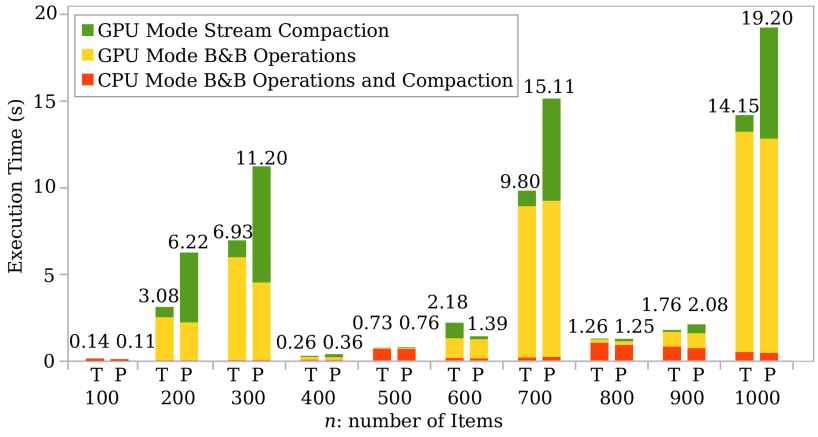

Fig. 7. Comparison and breakdown of execution time with different number of items. In this figure, the notations "T" and "P" refer to this work and previous work [9], respectively.

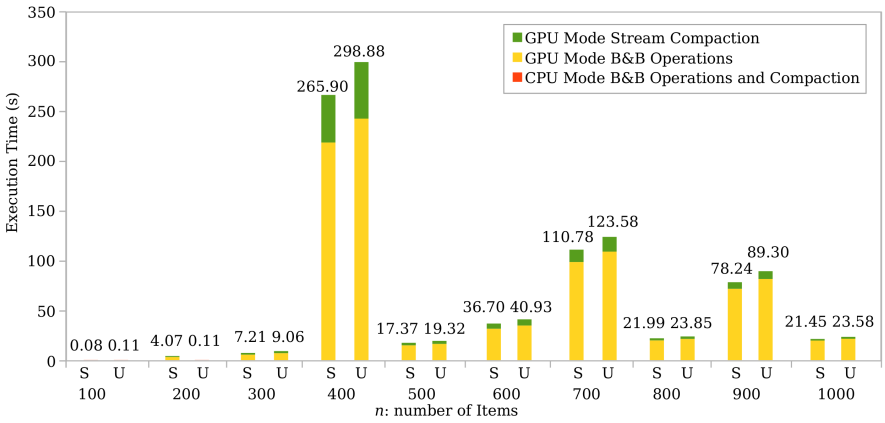

Fig. 8. Comparison and breakdown of execution time with different number of items. In this figure, the notations "S" and "U" refer to the separated and unified schemes, respectively.

5.3 Comparison of Separated and Unified Schemes

In this experiment, we allocated 1 GB (i.e., half) of GPU memory because the experimental machine 1 had only 2 GB GPU memory. Thus, we thought the efficiency of GPU memory usage would be more important. As for the CPU memory, the same amount (20 GB) was used for execution. We compared the execution time of both schemes by selecting a representative instance for each instance class in the same manner described in Sect. 5.2. As we metioned above, the buffer size of the unified scheme is 1.5 times larger than that of the separated scheme. Consequently, the unified scheme allowed 26.2 million subproblems to be processed at a time on the GPU, while the separated scheme allowed 17.4 million subproblems (Fig. 8).

According to the result, the execution time of GPU-based B&B operations (that are overlapped with data transfer) of the separated scheme was 12.3% shorter than that of the unified scheme. The execution time of stream compation of the separated scheme was 19.6% shorter than that of the unified scheme. Totally, the separated scheme was 13.1% faster than the unified scheme. We therefore concluded that the separated scheme is better than the unified scheme from the aspect of performance.

6 Conclusion

In this paper, we presented a GPU-accelerated, out-of-core B&B method for solving the 0-1 knapsack problem. The maximum problem size depends on the capacity of the CPU memory rather than that of the GPU memory. To realize this relaxation, our method buffers active subproblems in the CPU memory instead of the GPU memory. Because such a CPU-centric management scheme can

suffer from increased amount of data transfer between the CPU and GPU, our method minimizes the data transfer overhead by a stream compaction strategy and hides the minimized overhead by a double buffering strategy that overlaps data transfer with GPU-based B&B operations.

In our experiments, we found that our out-of-core method stored 33.7 times more subproblems at a time, solving twice more problems than a previous in-core method. We also found that our completely CPU-based compaction strategy was 4.5 times faster than the previous compaction strategy. Moreover, we determined that the separated scheme was better because the separated scheme ran 13.1% faster than the unified scheme.

Our future work includes investigation of more sophisticated ways to traverse the search tree, e.g., dynamically choosing breadth first search and depth first search, which improves robustness against rapidly growing subproblems.

Acknowledgments

This study was supported in part by the Japan Society for the Promotion of Science KAKENHI Grant Numbers 15H01687, 16H02801 and 15K12008. We are also grateful to the anonymous reviewers for their valuable comments.

References

1. S. Martello, P. Toth, Knapsack Problems: Algorithms and Computer Implementations, John Wiley & Sons Ltd., Baffins Lane, Chichester, 1990.
2. A. H. Land, A. G. Doig, An automatic method of solving discrete programming problems, Econometrica 28 (3) (1960) 497–520.
3. J. Lin, J. A. Storer, Processor-efficient hypercube algorithms for the knapsack problem, J. Parallel and Distributed Computing 13 (3) (1991) 332–337.
4. J. Eckstein, C. A. Phillips, W. E. Hart, Pico: An object-oriented framework for parallel branch and bound, Studies in Computational Mathematics 8 (2001) 219–265.
5. J.-P. Goux, S. Kulkarni, M. Yoder, J. Linderoth, Master-worker: An enabling framework for applications on the computational grid, Cluster Computing 4 (1) (2001) 63–70.
6. Y. Tanaka, M. Sato, M. Hirano, H. Nakada, S. Sekiguchi, Performance evaluation of a firewall-compliant Globus-based wide-area cluster system, in: Proc. HPDC'00, 2000, pp. 121–128.
7. V. Boyer, D. E. Baz, M. Elkihel, Solving knapsack problems on GPU, Computers & Operations Research 39 (1) (2012) 42–47.
8. A. Boukedjar, M. E. Lalami, D. El-Baz, Parallel branch and bound on a CPU-GPU system, in: Proc. PDP'12, 2012, pp. 392–398.
9. M. E. Lalami, D. El-Baz, GPU implementation of the branch and bound method for knapsack problems, in: Proc. IPDPSW'12, 2012, pp. 1769–1777.
10. M. Pedemonte, E. Alba, F. Luna, Towards the design of systolic genetic search, in: Proc. IPDPSW'12, 2012, pp. 1778–1786.
11. D. Luebke, G. Humphreys, How GPUs work, Computer 40 (2) (2007) 96–100.

12. F. Ino, Y. Munekawa, K. Hagihara, Sequence homology search using fine grained cycle sharing of idle GPUs, IEEE Trans. Parallel and Distributed Systems 23 (4) (2012) 751–759.
13. Y. Mitani, F. Ino, K. Hagihara, Parallelizing exact and approximate string matching via inclusive scan on a GPU. IEEE Transactions on Parallel and Distributed Systems (in print).
14. T. Carneiro, A. E. Muritiba, M. Negreiros, G. A. L. de Campos, A new parallel schema for branch-and-bound algorithms using GPGPU, in: Proc. SBAC-PAD'11, 2011, pp. 41–47.
15. G. B. Dantzig, Discrete variable extremum problems, Operations Research 5 (2) (1957) 266–277.
16. N. Bell, J. Hoberock, Thrust: A Productivity-Oriented Library for CUDA, Morgan Kaufmann, San Mateo, CA, 2011, Ch. 26, `http://thrust.github.io/`.
17. Steve Rennich, CUDA C/C++ Streams and Concurrency, Nvidia GTC express, 2011, `http://on-demand.gputechconf.com/gtc-express/2011/presentations/StreamsAndConcurrencyWebinar.pdf`.
18. S. Martello, D. Pisinger, P. Toth, New trends in exact algorithms for the 0-1 knapsack problem, European J. Operational Research 123 (2) (2000) 325–332.
19. S. Martello, D. Pisinger, P. Toth, Dynamic Programming and Tight Bounds for the 0-1 Knapsack Problem, Datalogisk Institut København: DIKU-Rapport, Datalogisk Institut, Københavns Universitet, 1997.
20. CUDA Toolkit Documentation, Nvidia, 2017, `http://docs.nvidia.com/cuda/index.html`.