

SPECIAL ISSUE PAPER

Efficient CPU-GPU cooperative computing for solving the
subset-sum problem[‡]

Lanjuan Wan^{1,2}, Kenli Li^{1,2,*,†}, Jing Liu^{1,2} and Keqin Li^{1,2,3}

¹College of Computer Science and Electronic Engineering, Hunan University, Changsha, Hunan 410082, China

²National Supercomputing Center in Changsha, Changsha, Hunan 410082, China

³Department of Computer Science, State University of New York, New Paltz, NY 12561, USA

SUMMARY

Heterogeneous CPU-GPU system is a powerful way to accelerate compute-intensive applications, such as the subset-sum problem. Many parallel algorithms for solving the problem have been implemented on graphics processing units (GPUs). However, these GPU implementations may fail to fully utilize all the CPU cores and the GPU resources. When the GPU performs computational task, only one CPU core is used to control the GPUs, and all the remaining CPU cores are in idle state, which leads to large amounts of available CPU resources being wasted. This paper proposes an efficient CPU-GPU cooperative computing scheme for solving the subset-sum problem, which enables the full utilization of all the computing power of both CPUs and GPUs. In order to find the most appropriate task distribution ratio between CPUs and GPUs, this paper establishes a simple but effective task distribution model. Considering the high CPU-GPU communication overhead and the unbalanced workload between CPUs and GPUs may greatly reduce the performance, an incremental data transfer method is proposed to reduce the CPU-GPU communication overhead, and a feedback-based dynamic task distribution scheme is designed to effectively balance the workload between CPUs and GPUs during runtime. The experimental results show that the CPU-GPU cooperative computing achieves a significant performance benefit over the CPU-only or GPU-only computing. Copyright © 2015 John Wiley & Sons, Ltd.

Received 16 April 2014; Revised 1 December 2014; Accepted 17 July 2015

KEY WORDS: CPU-GPU cooperative computing; CUDA; knapsack problem; parallel *two-list* algorithm; subset-sum problem

1. INTRODUCTION

Given n positive integers $W = [w_1, w_2, \dots, w_n]$ and a positive integer M , the subset-sum problem (SSP) is the problem of finding a set $I \subseteq \{1, 2, \dots, n\}$, such that $\sum w_i = M, i \in I$. In other words, the goal is to find a binary n -tuple solution $X = [x_1, x_2, \dots, x_n]$ for the equation

$$\sum_{i=1}^n w_i x_i = M, x_i \in \{0, 1\}. \quad (1)$$

Subset-sum problem is well-known to be non-deterministic polynomial-time complete (NP-complete) and it is a special case of the 0/1 knapsack problem. It has many real-life applications, such as capital budgeting, job scheduling, resource allocation, and project selection [1]. In recent

*Correspondence to: Kenli Li, College of Computer Science and Electronic Engineering, Hunan University, Hunan 410082, China.

†E-mail: lk1510@263.net

‡An earlier version of this paper was presented at the 2014 International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM'14).

decades, many exact and heuristic algorithms have been employed to solve SSP. A well-known sequential algorithm is the *two-list* algorithm proposed by Horowitz and Sahni [2], which solves SSP in time $O(n2^{n/2})$ with $O(2^{n/2})$ memory space. In order to effectively reduce the computation time of SSP, the parallelization of the *two-list* algorithm has been extensively discussed in [3–6]. In particular, Li *et al.* [6] proposed an optimal parallel *two-list* algorithm without memory conflict, which solves SSP in time $O(2^{n/4}(2^{n/4})^\varepsilon)$ with $O((2^{n/4})^{1-\varepsilon})$ processors and $O(2^{n/2})$ memory space, where $0 \leq \varepsilon \leq 1$.

Recently, heterogeneous CPU-GPU system has been widely used, which is a powerful way to deal with compute-intensive applications, because GPU can offer massive data parallelism and huge computational horsepower [7, 8]. A large effort has been performed to accelerate solving SSP by using the GPUs. Bokhari [9] explored a parallelization of the dynamic programming algorithm, which solves SSP on a 240-core NVIDIA FX 5800 GPU (NVIDIA Corporation, Santa Clara, CA, USA) via Compute Unified Device Architecture (CUDA). Wan *et al.* [10] implemented the parallel *two-list* algorithm of Li *et al.* on a GPU using CUDA, which successfully accelerates the computation of SSP. Some works [11–13] have also been made to accelerate solving the 0/1 knapsack problem on a GPU. These existing research works provide significant speedup over the traditional CPU-based implementation, however, they may fail to fully utilize all the CPU cores and the GPU resources at the same time. Specifically, when running a GPU application in a heterogeneous CPU-GPU system, only one CPU core is used to control the GPUs, and all the remaining CPU cores are in idle state while the GPU performs computational task, which leads to large amounts of available CPU resources being wasted. Therefore, how to find an effective method to make full use of the potential computational power of both CPUs and GPUs is very important.

The CPU-GPU cooperative computing has recently attracted the attention of many researchers and application developers. Some applications have been reported to successfully implement the CPU-GPU cooperative computing, instead of the CPU-only or GPU-only computing, such as matrix multiplication [14–16], fast Fourier transformation [17], LU factorization [18], QR factorization [19, 20], unsymmetric sparse linear system [21], radiation physics [22], molecular dynamics [23], conjugate gradient method [24], divide-and-conquer algorithm [25], and branch-and-bound algorithm [26]. These works show that the CPU-GPU cooperative computing has much better performance than the CPU-only or GPU-only computing. However, the efficient CPU-GPU cooperative computing for solving SSP remains a difficult challenge. The major difficulties are as follows: (1) how to assign the most suitable workload to both CPUs and GPUs to maximize the utilization of all computational resources; (2) how to minimize the CPU-GPU communication overhead, because the high CPU-GPU communication cost could be a performance bottleneck for some applications; and (3) how to achieve good load balancing between CPUs and GPUs, this is due to the load imbalance that significantly affects the full utilization of heterogeneous devices.

In this paper, based on our previous work [27] and on the optimal parallel *two-list* algorithm of Li *et al.* [6], we develop an efficient CPU-GPU cooperative computing scheme for solving SSP. However, the CPU-GPU cooperative implementation of the parallel *two-list* algorithm is not straightforward and needs to make a big effort. In order to find the most appropriate task distribution ratio between CPUs and GPUs, we propose a simple but effective task distribution model. Considering the data transfer and unbalanced workload between CPUs and GPUs may greatly reduce the overall performance, we propose an incremental data transfer method to reduce the CPU-GPU communication overhead and propose a feedback-based dynamic task distribution scheme to balance the workload between CPUs and GPUs during runtime. We conduct a series of experiments on three different hardware platforms. The results show that the CPU-GPU cooperative computing significantly outperforms the CPU-only or GPU-only computing. Our study makes the following contributions:

- A simple but effective task distribution model is established to find the most appropriate task distribution ratio between CPUs and GPUs, and the experimental results prove that the proposed model can find reasonable task distribution ratio.

- An incremental data transfer method is proposed to reduce the CPU-GPU communication overhead.
- A feedback-based dynamic task distribution scheme is designed to effectively balance the workload between CPUs and GPUs during runtime.
- A series of experiments are carried out to compare the performance of the CPU-GPU cooperative implementation with that of the best sequential implementation, the CPU-only implementation, and the GPU-only implementation.

The rest of this paper is organized as follows. Section 2 describes two different CPU-GPU cooperative computing methods and a task distribution model. Section 3 presents a CPU-GPU cooperative implementation of the parallel *two-list* algorithm. Section 4 provides two performance optimization schemes. Section 5 gives the experimental results and performance analysis. Section 6 concludes this paper and discusses the future work.

2. THE CPU-GPU COOPERATIVE COMPUTING ENVIRONMENT

In this section, we first describe two different CPU-GPU cooperative computing methods and then give a task distribution model.

2.1. The cooperative computing method

This section gives a brief description of two different CPU-GPU cooperative computing methods, which basically explains how to fully exploit the potential processing power of both CPUs and GPUs, as shown in Figure 1.

The main idea of the first CPU-GPU cooperative computing method is as follows: CPU thread 0 is dedicated to control and communicate with the GPUs, all the remaining available CPU threads together with all the available GPU threads to cooperatively perform the whole computational task. The typical flow of the first cooperative computing method is shown in Figure 1(a). Specifically, CPU thread 0 firstly transfers part of the input data from the host memory to the device memory, next, it invokes the CUDA kernel, then all GPU threads run the kernel in parallel, and finally, CPU thread 0 transfers the output data back to the host memory. At the same time, all the other CPU threads process the input data allocated to the CPUs in parallel.

The main idea of the second CPU-GPU cooperative computing method is similar to that of the first method. The typical flow of the second cooperative computing method is shown in Figure 1(b). Specifically, CPU thread 0 is dedicated to control the GPUs, after it invokes the CUDA kernel, all GPU threads run the kernel in parallel. In the meantime, CPU thread 1 firstly transfers part of the

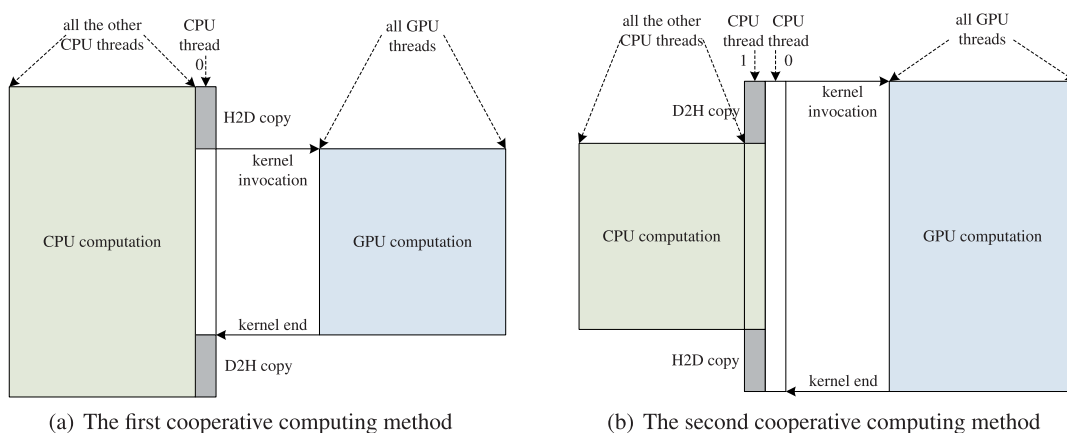


Figure 1. Two different CPU-GPU cooperative computing methods. (a) The first cooperative computing method and (b) the second cooperative computing method.

input data from the device memory to the host memory, then all the other CPU threads together with it to perform the computational task in parallel, and finally, CPU thread 1 transfers the output data from the host memory to the device memory.

The differences between the two methods are as follows: as for the first method, the data to be processed by the GPUs comes from the host memory, and the processed data may need to be copied back to the host memory; as for the second method, the data to be processed by the CPUs comes from the device memory, and the processed data may need to be copied back to the device memory. Note that the second method is employed in our proposed CPU-GPU cooperative implementation of the parallel *two-list* algorithm. The applications of this method are demonstrated in Algorithm 1 and Algorithm 7 proposed later in the paper. The next section explains why this method is adopted.

2.2. The task distribution model

This section describes a simple but effective task distribution model, which aims to find the most appropriate task distribution ratio between CPUs and GPUs.

The determination of the task distribution ratio needs to consider the following factors: processing capabilities and memory capacities of the host side and the device side, the bandwidth, the actual time to run the given program only on the CPUs or GPUs, and the CPU-GPU communication overhead. To facilitate our discussion, some notation used in this model is listed in Table I.

For both CPU-GPU cooperative computing methods described previously, the cooperative computing is considered finished only when both the CPUs and GPUs have finished their assigned workload; therefore, the total execution time of the CPU-GPU cooperative implementation is as follows:

$$T_{total} = \max(T'_{cpu}, T'_{gpu}). \quad (2)$$

For the first cooperative computing method, the CPU execution time of the CPU-GPU cooperative implementation can be calculated as follows:

$$T'_{cpu} = T_{cpu} \times R. \quad (3)$$

For the second cooperative computing method, T'_{cpu} can be calculated as follows:

$$T'_{cpu} = (T_{cpu} + T_{comm}) \times R. \quad (4)$$

The GPU execution time of the CPU-GPU cooperative implementation mainly consists of kernel execution time, GPU thread creation and destruction time, GPU synchronization time, and data transfer time. For the first cooperative computing method, T'_{gpu} can be calculated as follows:

$$T'_{gpu} = (T_{gpu} + T_{comm}) \times (1 - R). \quad (5)$$

Table I. Notation used in the task distribution model.

Notation	Description
D	the total workload
D_{cpu}	the workload assigned to the CPUs
D_{gpu}	the workload assigned to the GPUs
R	the proportion of the workload assigned to the CPUs
T_{cpu}	the running time of the CPU-only implementation
T_{gpu}	the running time of the GPU-only implementation
T'_{cpu}	the CPU execution time of the CPU-GPU cooperative implementation
T'_{gpu}	the GPU execution time of the CPU-GPU cooperative implementation
T_{comm}	the additional CPU-GPU communication time of the CPU-GPU cooperative implementation

For the second cooperative computing method, T'_{gpu} can be calculated as follows:

$$T'_{gpu} = T_{gpu} \times (1 - R). \quad (6)$$

In Equation (2), it is easy to see that the task distribution ratio can be considered optimal when $T'_{cpu} = T'_{gpu}$. For the first cooperative computing method, we can obtain the following equation:

$$T_{cpu} \times R = (T_{gpu} + T_{comm}) \times (1 - R). \quad (7)$$

By solving Equation (7), we obtain

$$R = \frac{T_{gpu} + T_{comm}}{T_{cpu} + T_{gpu} + T_{comm}}. \quad (8)$$

For the second cooperative computing method, we also can obtain the following equation:

$$(T_{cpu} + T_{comm}) \times R = T_{gpu} \times (1 - R). \quad (9)$$

By solving Equation (9), we obtain

$$R = \frac{T_{gpu}}{T_{cpu} + T_{gpu} + T_{comm}}. \quad (10)$$

For both cooperative computing methods, after the task distribution ratio has been determined, the workload assigned to the CPUs and GPUs can be calculated as follows:

$$\begin{cases} D_{cpu} = \lfloor D \times R \rfloor; \\ D_{gpu} = D - D_{cpu}. \end{cases} \quad (11)$$

For a specific problem, assuming that T_{comm} is more or less the same for two different cooperative computing methods. If $T_{cpu} < T_{gpu}$, it is not hard to observe that the first method can give better performance than the second method. If $T_{cpu} = T_{gpu}$, both methods can achieve similar performance. If $T_{cpu} > T_{gpu}$, the second method outperforms the first method. For the parallel *two-list* algorithm, the experimental results show that $T_{cpu} > T_{gpu}$, so we adopt the second method.

Although the task distribution model described earlier could effectively find the appropriate task distribution ratio for the parallel *two-list* algorithm, it is based on the assumptions that the problem being solved has good scalability and the problem size is fixed. Therefore, it would not hold for all different types of parallel applications, and those equations used for calculating the task distribution ratio are not generic.

3. THE PROPOSED CPU-GPU COOPERATIVE IMPLEMENTATION

In this section, we first briefly introduce the parallel *two-list* algorithm for solving SSP and then focus on the CPU-GPU cooperative implementation of the algorithm.

3.1. The parallel *two-list* algorithm

Based on the Single Instruction Multiple Data model with shared memory, Li *et al.* [6] proposed an optimal parallel *two-list* algorithm, which contains three stages as follows.

The parallel generation stage, which is designed for generating two sorted lists. Specifically, we first obtain an n -element input vector W and divide it into two equal parts: $W_1 = [w_1, w_2, \dots, w_{n/2}]$ and $W_2 = [w_{n/2+1}, w_{n/2+2}, \dots, w_n]$. Secondly, we use k threads to concurrently produce all $2^{n/2}$ possible subset sums of W_1 , and store them into the list $A = [a_1, a_2, \dots, a_{2^{n/2}}]$ in nondecreasing order, where k is the number of required threads and it is less than or equal to the maximum number of threads available. Finally, we use k threads to concurrently produce all $2^{n/2}$ possible subset sums of W_2 , and store them into the list $B = [b_1, b_2, \dots, b_{2^{n/2}}]$ in nonincreasing order.

The parallel pruning stage, which is used to shrink the search space of each thread. In short, we first evenly divide lists A and B into k blocks respectively. For clarity, let $A = [\overline{A_1}, \overline{A_2}, \dots, \overline{A_i}, \dots, \overline{A_k}]$ and $B = [\overline{B_1}, \overline{B_2}, \dots, \overline{B_j}, \dots, \overline{B_k}]$, where $\overline{A_i} = [\overline{a_{i,1}}, \overline{a_{i,2}}, \dots, \overline{a_{i,2^{n/2}/k}}]$, $\overline{B_j} = [\overline{b_{j,1}}, \overline{b_{j,2}}, \dots, \overline{b_{j,2^{n/2}/k}}]$ and $1 \leq i, j \leq k$. Each element $\overline{a_{i,r}}$ in the sublist $\overline{A_i}$ represents a subset sum of A , where $1 \leq r \leq 2^{n/2}/k$. Secondly, the block $\overline{A_i}$ and the entire list B are assigned to the thread P_i , where $1 \leq i \leq k$. Finally, we use the prune rule [6] to shrink the search space of each thread. Before pruning, the number of block pairs is k^2 . After pruning, the number of the picked block pairs is at most $2k - 1$.

The parallel search stage, which is constructed to find a solution from the combinations of the two sorted lists. Briefly, those picked block pairs are evenly assigned to k threads; it is easy to see that each thread will search two block pairs at most. For the sake of discussion, assuming that the block pair $(\overline{A_s}, \overline{B_t})$ is assigned to the thread P_i , where $\overline{A_s} = [\overline{a_{s,1}}, \overline{a_{s,2}}, \dots, \overline{a_{s,2^{n/2}/k}}]$, $\overline{B_t} = [\overline{b_{t,1}}, \overline{b_{t,2}}, \dots, \overline{b_{t,2^{n/2}/k}}]$, and $1 \leq i, s, t \leq k$. The thread P_i finds the top elements of the block pair $(\overline{A_s}, \overline{B_t})$, if $\overline{a_{s,1}} + \overline{b_{t,1}} = M$, implying that a solution is found; if $\overline{a_{s,1}} + \overline{b_{t,1}} < M$, then P_i continues to search the next element of the block $\overline{A_s}$; otherwise, P_i continues to search the next element of the block $\overline{B_t}$. The aforementioned process is repeated until the last element of the block $\overline{A_s}$ or $\overline{B_t}$ has been retrieved.

3.2. The cooperative implementation of the generation stage

The cooperative implementation of the generation stage begins with the initialization on the host and device sides. Firstly, we get an n -element input vector W and divide it into two equal parts, W_1 and W_2 on the host side. Secondly, we allocate device memory for W_1 and W_2 on the device side, and copy W_1 and W_2 to the device memory. Thirdly, we initialize the knapsack capacity M on the host side, allocate constant memory for M on the device side, and copy M to the constant memory.

Table II. Notation used in the generation stage.

Notation	Description
A_i	a list which resides in the host memory, $A_i = [a_{i,1}, a_{i,2}, \dots, a_{i,2^i}]$, where $1 \leq i \leq n/2$
$a_{i,r}$	a subset sum, where $1 \leq r \leq 2^i$
$A_i[p..q]$	a list which resides in the host memory, $A_i[p..q] = [a_{i,p}, a_{i,p+1}, \dots, a_{i,q}]$, where $1 \leq p, q \leq 2^i$
d_A_i	a list which resides in the device memory, $d_A_i = [a_{i,1}, a_{i,2}, \dots, a_{i,2^i}]$, where $1 \leq i \leq n/2$
A_i^1	a list which resides in the host memory, $A_i^1 = [a_{i,1} + w_{i+1}, a_{i,2} + w_{i+1}, \dots, a_{i,2^i} + w_{i+1}]$
$d_A_i^1$	a list which resides in the device memory, $d_A_i^1 = [a_{i,1} + w_{i+1}, a_{i,2} + w_{i+1}, \dots, a_{i,2^i} + w_{i+1}]$
D_1	the total number of elements in d_A_i during the add item process
D_{cpu1}	the number of elements assigned to the CPUs in d_A_i during the add item process
D_{gpu1}	the number of elements assigned to the GPUs in d_A_i during the add item process
D_{cpu1}^1	the number of elements assigned to the CPUs in $d_A_i^1$ during the add item process
D_{gpu1}^1	the number of elements assigned to the GPUs in $d_A_i^1$ during the add item process
R_1	the task distribution ratio used during the add item process
k_{cpu1}	the number of required CPU threads during the add item process
k_{gpu1}	the number of required GPU threads during the add item process
D_2	the total number of elements in d_A_i during the partition and merge processes
D_{cpu2}	the number of elements assigned to the CPUs in d_A_i during the partition and merge processes
D_{gpu2}	the number of elements assigned to the GPUs in d_A_i during the partition and merge processes
D_{cpu2}^1	the number of elements assigned to the CPUs in $d_A_i^1$ during the partition and merge processes
D_{gpu2}^1	the number of elements assigned to the GPUs in $d_A_i^1$ during the partition and merge processes
R_2	the task distribution ratio used during the partition and merge processes
k_{cpu2}	the number of required CPU threads during the partition and merge processes
k_{gpu2}	the number of required GPU threads during the partition and merge processes
c_{max}	the maximum number of CPU threads available
g_{max}	the maximum number of GPU threads available

Fourthly, we allocate device memory for lists d_A and d_B on the device side. Here, the list d_A is used to store all $2^{n/2}$ subset sums of W_1 , and the list d_B is used to store all $2^{n/2}$ subset sums of W_2 . Fifthly, we initialize $d_A_1 = [0, w_1]$ and $d_B_1 = [w_{n/2+1}, 0]$. Note that d_A_i represents a list, which resides in the device memory, $d_A_i = [a_{i,1}, a_{i,2}, \dots, a_{i,2^i}]$, where $1 \leq i \leq n/2$. Each element $a_{i,r}$ in the list d_A_i represents a subset sum, where $1 \leq r \leq 2^i$. The same convention is used for the list d_B_i . Finally, we allocate host memory for lists A and B on the host side, and use A and B to store the subset sums of W_1 and W_2 , respectively.

After the initialization phase has been completed, we will describe the generation procedure of the nondecreasing list d_A . The generation procedure of the nonincreasing list d_B is almost the same as that of list d_A , we do not discuss it here for the sake of brevity. Some notation used in this generation stage is summarized in Table II.

3.2.1. The cooperative computing scheme of the generation stage. The whole process of generating the list d_A needs to execute $n/2 - 1$ iterations to complete. Each iteration is made up of the following sequential activities: adding item, partitioning, and merging. The cooperative computing scheme of the generation stage is described in Algorithm 1.

The observations for the $(i-1)$ -th iteration of the generation procedure show that T_{gpu_i} is far less than $T_{cpu_i} + T_{comm_i}$ when i is small, where $2 \leq i \leq n/2$. This means that the cooperative computing is not suitable for small computational task. Therefore, we introduce a threshold λ to decide whether the computational task should be performed using GPU-only or using both CPUs and GPUs. The threshold value can be determined through the following experiments: first, we run the generation procedure on the CPUs to obtain T_{cpu_i} (the CPU execution time of the $(i-1)$ -th iteration); secondly, we run it on the GPUs to obtain T_{gpu_i} (the GPU execution time of the $(i-1)$ -th iteration); thirdly, we run it on both CPUs and GPUs to obtain T_{comm_i} (the CPU-GPU communication time of the $(i-1)$ -th iteration), where $R_1=1$ and $R_2=1$; and finally, the threshold value is determined according to T_{cpu_i} , T_{gpu_i} and T_{comm_i} , if T_{gpu_i} is far less than $T_{cpu_i} + T_{comm_i}$, then $\lambda = i$, where $2 \leq i \leq n/2$.

When $2 \leq i \leq \lambda - 1$, we only use the GPUs to perform the generation procedure. Specifically, during the $(i-1)$ -th iteration, we first use k_{gpu1} GPU threads to add the item w_i to each element of the list d_A_{i-1} in parallel, generating a new list $d_A_{i-1}^1 = [a_{i-1,1} + w_i, a_{i-1,2} + w_i, \dots, a_{i-1,2^{i-1}} + w_i]$, where $k_{gpu1} = \min(g_{\max}, 2^{i-1})$. Then, we use k_{gpu2} GPU threads to merge lists d_A_{i-1} and $d_A_{i-1}^1$ into a new nondecreasing list d_A_i in parallel, where $k_{gpu2} = \min(g_{\max}, 2^{i-1})$.

When $\lambda \leq i \leq n/2$, we use both CPUs and GPUs to cooperatively perform the generation procedure. It is easy to observe that each iteration consists of the following six steps:

- Step 1: Determine the workload of the CPUs and GPUs, respectively, according to the task distribution ratio R_1 obtained from Equation (10) during the add item process.
- Step 2: Determine the number of required CPU and GPU threads during the add item process.
- Step 3: Execute the add item process on both the host and device sides simultaneously.
- Step 4: Determine the workload of the CPUs and the GPUs, respectively, according to the task distribution ratio R_2 obtained from Equation (10) during the partition and merge processes.
- Step 5: Determine the number of required CPU and GPU threads during the partition and merge processes.
- Step 6: Execute the partition and merge processes on both the host and device sides simultaneously.

As can be seen from Algorithm 1, two concurrent CPU threads are used to control the cooperative execution of the add item process. CPU thread 0 is used to control the GPUs, and k_{gpu1} GPU threads are used to concurrently execute the add item process on the device side. CPU thread 1 is used to control the CPUs. By exploiting the nested parallelism in OpenMP, k_{cpu1} CPU threads are used to concurrently perform the add item process on the host side. Similarly, we use two concurrent CPU threads to control the cooperative execution of the partition and merge processes. In addition, we can use CUDA stream to achieve the overlap of data transfer and kernel execution, i.e., simultaneously execute a CUDA kernel while performing the data transfer between CPUs and GPUs.

Algorithm 1 The cooperative computing scheme of the generation stage

Require: $W_1 = [w_1, w_2, \dots, w_{n/2}]$, $d_{-A_1} = [0, w_1]$, R_1, R_2

```

1: for  $i = 2$  to  $\lambda - 1$  do
2:   use  $k_{gpu1} = \min(g_{max}, 2^{i-1})$  GPU threads to perform the add item process in parallel;
3:   use  $k_{gpu2} = \min(g_{max}, 2^{i-1})$  GPU threads to perform the partition and merge processes in parallel;
4: end for
5: for  $i = \lambda$  to  $n/2$  do
6:    $\triangleright$  the add item process
7:    $D_1 = 2^{i-1}$ ,  $D_{cpu1} = \lfloor D_1 \times R_1 \rfloor$ ,  $D_{cpu1}^1 = \lfloor D_1 \times R_1 \rfloor$ ,  $D_{gpu1} = D_1 - D_{cpu1}$ ,  $D_{gpu1}^1 = D_1 - D_{cpu1}^1$ ;
8:    $k_{cpu1} = \min(c_{max}, D_{cpu1})$ ,  $k_{gpu1} = \min(g_{max}, D_{gpu1})$ ;
9:   omp_set_nested(1);
10:  #pragma omp parallel num_threads(2) {
11:    omp_tid = omp_get_thread_num();
12:    if omp_tid = 0 then
13:      for all  $k_{gpu1}$  GPU threads do in parallel
14:        produce new list  $d_{-A_{i-1}}^1[1..D_{gpu1}^1]$  by adding the item  $w_i$  to each element of  $d_{-A_{i-1}}[1..D_{gpu1}]$ ;
15:      end for
16:    else
17:      copy each element of  $d_{-A_{i-1}}[D_{gpu1} + 1..D_1]$  to  $A_{i-1}[D_{gpu1} + 1..D_1]$ ;
18:      for all  $k_{cpu1}$  CPU threads do in parallel
19:        produce new list  $A_{i-1}^1[D_{gpu1}^1 + 1..D_1]$  by adding  $w_i$  to each element of  $A_{i-1}[D_{gpu1}^1 + 1..D_1]$ ;
20:      end for
21:      copy each element of  $A_{i-1}^1[D_{gpu1}^1 + 1..D_1]$  to  $d_{-A_{i-1}}^1[D_{gpu1} + 1..D_1]$ ;
22:    end if }
23:   $\triangleright$  the partition and merge processes
24:   $D_2 = 2^{i-1}$ ,  $D_{cpu2} = \lfloor D_2 \times R_2 \rfloor$ ,  $D_{gpu2} = D_2 - D_{cpu2}$ ;
25:  perform Algorithm 2 to get  $D_{gpu2}^1$ ,  $D_{cpu2}^1 = D_2 - D_{gpu2}^1$ ;
26:   $k_{cpu2} = \min(c_{max}, 2^{\lfloor \log(D_{cpu2} + D_{cpu2}^1) \rfloor})$ ,  $k_{gpu2} = \min(g_{max}, 2^{\lfloor \log(D_{gpu2} + D_{gpu2}^1) \rfloor})$ ;
27:  #pragma omp parallel num_threads(2) {
28:    omp_tid = omp_get_thread_num();
29:    if omp_tid = 0 then
30:      for all  $k_{gpu2}$  GPU threads do in parallel
31:        use the optimal parallel merging algorithm to merge  $d_{-A_{i-1}}[1..D_{gpu2}]$  and  $d_{-A_{i-1}}^1[1..D_{gpu2}^1]$  into  $d_{-A_i}[1..D_{gpu2} + D_{gpu2}^1]$ ;
32:      end for
33:    else
34:      copy each element of  $d_{-A_{i-1}}[D_{gpu2} + 1..D_2]$  to  $A_{i-1}[D_{gpu2} + 1..D_2]$ ;
35:      copy each element of  $d_{-A_{i-1}}^1[D_{gpu2}^1 + 1..D_2]$  to  $A_{i-1}^1[D_{gpu2}^1 + 1..D_2]$ ;
36:      for all  $k_{cpu2}$  CPU threads do in parallel
37:        use the optimal parallel merging algorithm to merge  $A_{i-1}[D_{gpu2} + 1..D_2]$  and  $A_{i-1}^1[D_{gpu2}^1 + 1..D_2]$  into  $A_i[D_{gpu2} + D_{gpu2}^1 + 1..2 \times D_2]$ ;
38:      end for
39:      copy each element of  $A_i[D_{gpu2} + D_{gpu2}^1 + 1..2 \times D_2]$  to  $d_{-A_i}[D_{gpu2} + D_{gpu2}^1 + 1..2 \times D_2]$ ;
40:    end if }
41:  end for
42:  return  $d_{-A_{n/2}}$ 

```

Algorithm 2 Determine the number of elements assigned to the GPUs in the list $d_{-A_{i-1}^1}$

Require: $d_{-A_{i-1}}, d_{-A_{i-1}^1}, D_{gpu2}, D_2$

- 1: $left = 1, right = D_2, D_{gpu2}^1 = 0;$
- 2: **while** $left \leq right$ **do**
- 3: $middle = (left + right)/2;$
- 4: **if** $d_{-A_{i-1}^1}[middle] = d_{-A_{i-1}}[D_{gpu2}]$ **then** $D_{gpu2}^1 = middle;$ **break;**
- 5: **else if** $d_{-A_{i-1}^1}[middle] < d_{-A_{i-1}}[D_{gpu2}]$ **then** $left = middle + 1;$
- 6: **else** $right = middle - 1;$ **end if**
- 7: **end while**
- 8: **if** $D_{gpu2}^1 = 0$ **then**
- 9: **if** $d_{-A_{i-1}^1}[middle] < d_{-A_{i-1}}[D_{gpu2}]$ **then** $D_{gpu2}^1 = middle;$
- 10: **else** $D_{gpu2}^1 = middle - 1;$ **end if**
- 11: **end if**
- 12: **return** D_{gpu2}^1

3.2.2. *The implementation of the add item process.* This section describes the CPU-GPU cooperative implementation of the add item process.

Algorithm 3 describes the add item kernel implemented with CUDA, which can be executed concurrently on the device side. We use k_{gpu1} GPU threads to run the add item kernel in parallel. The tid -th GPU thread adds the item w_i to the tid -th element of the list $d_{-A_{i-1}}$, generating the corresponding tid -th element of the list $d_{-A_{i-1}^1}$, where $1 \leq tid \leq D_{gpu1}$.

Algorithms 4 describes the add item subroutine implemented with Open Multi-Processing (OpenMP), which can be executed concurrently on the host side. We use k_{cpu1} CPU threads to add the item w_i to each element of the list $A_{i-1}[D_{gpu1} + 1..D_1]$ in parallel, generating a new list $A_{i-1}^1 = [D_{gpu1} + 1..D_1]$.

Algorithm 3 The add item kernel implemented with CUDA

Require: $w_i, d_{-A_{i-1}}, d_{-A_{i-1}^1}, D_{gpu1}$

- 1: $tid = blockDim.x * blockIdx.x + threadIdx.x + 1;$
- 2: **while** $tid \leq D_{gpu1}$ **do**
- 3: $d_{-A_{i-1}^1}[tid] = d_{-A_{i-1}}[tid] + w_i;$
- 4: $tid += blockDim.x * gridDim.x;$
- 5: **end while**
- 6: **return** $d_{-A_{i-1}^1}[1..D_{gpu1}]$

Algorithm 4 The add item subroutine implemented with OpenMP

Require: $w_i, A_{i-1}, A_{i-1}^1, k_{cpu1}, D_{gpu1}, D_1$

- 1: $\#pragma omp parallel for num_threads(k_{cpu1});$
- 2: **for** $j = D_{gpu1} + 1$ **to** D_1 **do**
- 3: $A_{i-1}^1[j] = A_{i-1}[j] + w_i;$
- 4: **end for**
- 5: **return** $A_{i-1}^1[D_{gpu1} + 1..D_1]$

3.2.3. *The implementation of the partition and merge processes.* The implementation of the partition and merge processes on the CPUs using OpenMP is almost the same as that on the GPUs using CUDA, we only introduce the latter in this section.

The partition and merge processes use the optimal parallel merging algorithm presented in [28], which adopts a recursive *divide-and-conquer* strategy. Because the support for recursions in NVIDIA GPUs with compute capability less than 3.5 is weak, we have to use iteration instead

of recursion. A vector-based iterative implementation mechanism is developed in our previous work [10]. Currently, the new NVIDIA GPUs with compute capability 3.5 or higher support for a technology called ‘dynamic parallelism’, which allows a CUDA kernel to create and synchronize with new work directly on the GPUs. It can make GPU programming easier, particularly for algorithms traditionally considered difficult for GPUs such as recursive *divide-and-conquer* problems. Here, the dynamic parallelism is adopted in the implementation of the partition process.

On the device side, the whole partition process needs to execute $\log k_{gpu2} - 1$ recursion steps to complete. At the j -th recursion, we use 2^j GPU threads to execute the partition kernel described in Algorithm 5 in parallel, where $1 \leq j \leq \log k_{gpu2} - 1$. In order to store the partition information (i.e., median pair of sublists X and Y) generated by each GPU thread in each recursion, we declare a vector d_H whose size is $k_{gpu2} + 1$ in the device memory. Each element of d_H is a struct with two members u and v . We initialize the vector d_H as follows: $d_H[1].u = 1$, $d_H[1].v = 1$, $d_H[k_{gpu2} + 1].u = D_{gpu2}$, $d_H[k_{gpu2} + 1].v = D_{gpu2}^1$.

Algorithm 5 The partition kernel implemented with CUDA

```

1: __global__ void partitionKernel(int lowA, int highA, int lowB, int highB, int low, int high, int
   depth) {
2:   a = lowA, b = highA, c = lowB, d = highB;
3:   find the median pair (e, f) of dAi-1[a..b] and dAi-11[c..d] using the selection algorithm
   presented in [28];
4:   dH[(low + high + 1)/2].u = e;
5:   dH[(low + high + 1)/2].v = f;
6:   if depth = log k or depth > maxDepth then return;
7:   else
8:     create two CUDA streams: s1, s2;
9:     partitionKernel<<< 1, 1, 0, s1 >>> (lowA, e, lowB, f, low, (low + high + 1)/2, depth + 1);
10:    partitionKernel<<< 1, 1, 0, s2 >>> (e + 1, highA, f + 1, highB, (low + high + 1)/2, high,
   depth + 1);
11:    destroy two CUDA streams: s1, s2;
12:   end if }

```

In Algorithm 5, the partition kernel has seven parameters: low_A , $high_A$, low_B , $high_B$, low , $high$, and $depth$. The parameters low_A and $high_A$ represent the lower bound position and the upper bound position of sublist X , respectively. The parameters low_B and $high_B$ denote the lower bound position and the upper bound position of sublist Y , respectively. The parameters low and $high$ determine which element of d_H will be used to store the partition information. The parameter $depth$ refers to the depth of the recursion, whereas $maxDepth$ represents the GPU-supported maximum recursion depth. Before launching the partition kernel, whose seven parameters are specified as follows: $low_A = 1$, $high_A = D_{gpu2}$, $low_B = 1$, $high_B = D_{gpu2}^1$, $low = 1$, $high = k_{gpu2}$ and $depth = 1$. The calculation of the three parameters D_{gpu2} , D_{gpu2}^1 and k_{gpu2} is shown in lines 24–26 of Algorithm 1.

After completing the partition process on the device side, we use k_{gpu2} GPU threads to run the merge kernel described in Algorithm 6 in parallel. At first, the current GPU thread P_{tid} obtains the

Algorithm 6 The merge kernel implemented with CUDA

Require: $d_{A_{i-1}}$, $d_{A_{i-1}}^1$, d_H , k_{gpu2}

```

1: tid = blockIdx.x × blockDim.x + threadIdx.x + 1;
2: while tid ≤ kgpu2 do
3:   a = dH[tid].u, b = dH[tid].v, c = dH[tid + 1].u, d = dH[tid + 1].v;
4:   merge dAi-1[a..b] and dAi-11[c..d] into dAi[a + c - 1..b + d];
5:   tid += blockDim.x × gridDim.x;
6: end while
7: return dAi

```

Table III. Notation used in the pruning and search stages.

Notation	Description
D_3	the total number of elements in list d_A
D_{cpu3}	the number of elements assigned to the CPUs in list d_A
D_{gpu3}	the number of elements assigned to the GPUs in list d_A
R_3	the task distribution ratio used during the pruning and search stages
k_{cpu3}	the number of required CPU threads during the pruning and search stages
k_{gpu3}	the number of required GPU threads during the pruning and search stages
$isFound_cpu$	the variable used to determine whether a solution of SSP is found on the host side
$isFound_gpu_d$	the variable used to determine whether a solution of SSP is found on the device side
$isFound_gpu_h$	the variable used to store the value of $isFound_gpu_d$ on the host side

SSP, subset-sum problem.

partition information (a, b, c, d) from the vector d_H , where $1 \leq tid \leq k_{gpu2}$. Then, P_{tid} merges lists $d_A_{i-1}[a..b]$ and $d_A_{i-1}^1[c..d]$ into a new list $d_A_i[a + c - 1..b + d]$.

3.3. The cooperative implementation of the pruning and search stages

This section describes how to implement the pruning and search stages in the CPU-GPU cooperative computing environment. Some notation used in these two stages is illustrated in Table III.

3.3.1. The cooperative computing scheme of the pruning and search stages. Algorithm 7 describes the cooperative computing scheme of the pruning and search stages, which consists of the following six steps:

Algorithm 7 The cooperative computing scheme of the pruning and search stages

Require: d_A, d_B, A, B, R_3, M

```

1:  $D_3 = 2^{n/2}$ ,  $D_{cpu3} = \lfloor D_3 \times R_3 \rfloor$ ,  $D_{gpu3} = D_3 - D_{cpu3}$ ;
2:  $k_{cpu3} = \min(c_{max}, 2^{\lceil \log D_{cpu3}/2 \rceil})$ ,  $k_{gpu3} = \min(g_{max}, 2^{\lceil \log D_{gpu3}/2 \rceil})$ ;
3: omp_set_nested(1);
4: #pragma omp parallel num_threads(2) {
5:   omp_tid = omp_get_thread_num();
6:   if omp_tid = 0 then
7:     use  $k_{gpu3}$  GPU threads to perform the pruning routine in parallel on the device side;
8:     copy the value of isFound_gpu_d to isFound_gpu_h;
9:   else
10:    copy each element of  $d\_A[D_{gpu3} + 1..D_3]$  to  $A[D_{gpu3} + 1..D_3]$ , and copy all elements
    of  $d\_B$  to  $B$ ;
11:    use  $k_{cpu3}$  CPU threads to perform the pruning routine in parallel on the host side;
12:   end if }
13: if isFound_cpu = 1 or isFound_gpu_h = 1 then return the solution of SSP;
14: else
15:   #pragma omp parallel num_threads(2) {
16:     omp_tid = omp_get_thread_num();
17:     if omp_tid = 0 then
18:       use  $k_{gpu3}$  GPU threads to perform the search routine in parallel on the device side;
19:       copy the value of isFound_gpu_d to isFound_gpu_h;
20:     else use  $k_{cpu3}$  CPU threads to perform the search routine in parallel on the host side; end
     if }
21:   end if
22: if isFound_cpu = 1 or isFound_gpu_h = 1 then return the solution of SSP;
23: else return NULL; ▷ there is no solution end if

```

- Step 1: Determine the workload of the CPUs and GPUs, respectively, according to the task distribution ratio R_3 obtained from Equation (10). Note that all $2^{n/2}$ elements of the list d_B need to be assigned to the CPUs and GPUs, respectively.
- Step 2: Determine the number of required CPU and GPU threads.
- Step 3: Execute the pruning stage on both the host and device sides simultaneously. If a solution is found on the host side, then $isFound_cpu = 1$. If a solution is found on the device side, then $isFound_gpu_d = 1$.
- Step 4: Determine whether to carry out the next search stage according to the values of $isFound_cpu$ and $isFound_gpu_h$.
- Step 5: Execute the search stage on both the host and device sides simultaneously.
- Step 6: Output the final search results.

Because the pruning and search stages are implemented on the CPUs using OpenMP and that are implemented on the GPUs using CUDA are similar, we only introduce the latter in the next sections.

3.3.2. The implementation of the pruning stage. This section describes how to shrink the search space by using k_{gpu3} GPU threads to run the pruning kernel described in Algorithm 8 in parallel.

Before the pruning kernel is executed, we first evenly divide $d_A[1..D_{gpu3}]$ and d_B into k_{gpu3} blocks. Each block of d_A contains $blockA_{gpu} = \lfloor D_{gpu3}/k_{gpu3} \rfloor$ elements, and each block of d_B contains $blockB_{gpu} = \lfloor 2^{n/2}/k_{gpu3} \rfloor$ elements. Let $d_A[1..D_{gpu3}] = [d_A_1, \dots, d_A_i, \dots, d_A_{k_{gpu3}}]$ and $d_B = [d_B_1, \dots, d_B_j, \dots, d_B_{k_{gpu3}}]$, where $d_A_i = [a_{i,1}, a_{i,2}, \dots, a_{i,blockA_{gpu}}]$ and $d_B_j = [b_{j,1}, b_{j,2}, \dots, b_{j,blockB_{gpu}}]$. Secondly, we assign the block d_A_i and the entire list d_B to the GPU thread P_i , where $1 \leq i \leq k_{gpu3}$. Thirdly, we declare a vector d_S whose size is $2k_{gpu3} - 1$ in the device memory. Each element of d_S is a struct with two members $bidA$ and $bidB$, which are used to record the picked block index within d_A and d_B , respectively. Finally, we declare a variable $d_numOfPicked$ in the device memory, which is used to count the number of block pairs to be picked.

After the pruning kernel has been completed, if $isFound_gpu_d = 1$, we copy the solution back to the host memory and output it.

Algorithm 8 The pruning kernel implemented with CUDA

Require: $d_A, d_B, k_{gpu3}, blockA_{gpu}, blockB_{gpu}, M$

```

1:  $i = blockIdx.x \times blockDim.x + threadIdx.x + 1$ ;
2: while  $i \leq k_{gpu3}$  do
3:   for  $j = 1$  to  $k_{gpu3}$  do
4:     if  $isFound\_gpu\_d = 1$  then break; ▷ a solution is found
5:     else
6:        $X = a_{i,1} + b_{j,blockB_{gpu}}, Y = a_{i,blockA_{gpu}} + b_{j,1}$ ;
7:       if  $X = M$  or  $Y = M$  then  $atomicExch(\&isFound\_gpu\_d, 1)$ ;
8:       else if  $X < M$  and  $Y > M$  then
9:          $atomicAdd(\&d\_numOfPicked, 1)$ ;
10:         $d\_S[d\_numOfPicked].bidA = i, d\_S[d\_numOfPicked].bidB = j$ ;
11:       end if
12:     end if
13:   end for
14:    $i += blockDim.x \times gridDim.x$ ;
15: end while
16: return a solution or  $d\_S$ 

```

3.3.3. The implementation of the search stage. This section describes how to search a solution of SSP by using k_{gpu3} GPU threads to perform the search kernel described in Algorithm 9 in parallel.

Before the search kernel is executed, we evenly assign those picked block pairs to k_{gpu3} GPU threads. Because at most $2k_{gpu3} - 1$ block pairs are picked during the pruning stage, each thread will

search two block pairs at most. For clarity, assuming that the block pair $(\overline{d_{-A_s}}, \overline{d_{-B_t}})$ is assigned to the thread P_i , where $1 \leq i, s, t \leq k_{gpu3}$. During the search kernel execution, P_i finds the top elements of $(\overline{d_{-A_s}}, \overline{d_{-B_t}})$, if $\overline{a_{s,1}} + \overline{b_{t,1}} = M$, meaning that a solution is found; otherwise, P_i continues to search the next element of $\overline{d_{-A_s}}$ or $\overline{d_{-B_t}}$. The search process is repeated until the last element of $\overline{d_{-A_s}}$ or $\overline{d_{-B_t}}$ has been retrieved. After the search kernel has been completed, if $isFound_gpu_d = 1$, we copy the solution back to the host memory and output it.

Algorithm 9 The search kernel implemented with CUDA

Require: $d_A, d_B, d_S, d_numOfPicked, M$

```

1:  $i = blockIdx.x \times blockDim.x + threadIdx.x + 1$ ;
2: while  $i \leq d\_numOfPicked$  do
3:    $s = d\_S[i].bidA, t = d\_S[i].bidB, x = 1, y = 1$ ;
4:   while  $x \leq blockA_{gpu}$  and  $y \leq blockB_{gpu}$  do
5:     if  $isFound\_gpu\_d = 1$  then break; ▷ a solution is found
6:     else
7:       if  $\overline{a_{s,x}} + \overline{b_{t,y}} = M$  then  $atomicExch(\&isFound\_gpu\_d, 1)$ ;
8:       else if  $\overline{a_{s,x}} + \overline{b_{t,y}} < M$  then  $x = x + 1$ ;
9:       else  $y = y + 1$ ; end if
10:    end if
11:  end while
12:   $i += blockDim.x \times gridDim.x$ ;
13: end while
14: return a solution or NULL

```

4. PERFORMANCE OPTIMIZATION

The data transfer and unbalanced workload between CPUs and GPUs are two main obstacles to the performance of our proposed CPU-GPU cooperative implementation. In this section, we will show how to achieve better performance by addressing these two obstacles.

4.1. The incremental data transfer method

In our proposed CPU-GPU cooperative implementation, the high CPU-GPU communication overhead greatly affects the overall performance. An effective way to reduce the CPU-GPU communication overhead is to reduce the repeated data transfers between CPUs and GPUs, namely, to maximize the reuse of the data that resides in the host memory or device memory. In this section, we propose an incremental data transfer method to reduce the repeated data transfers.

Firstly, we describe the use of the incremental data transfer method during the generation stage. Algorithm 10 shows the improved generation stage using the method. When $2 \leq i \leq \lambda - 1$, because we only use the GPUs to perform the generation procedure, the method does not need to be considered. When $\lambda \leq i \leq n/2$, in each iteration, the use of the method consists of the following two steps:

Step 1: Before the add item process is executed, we first get $D_{cpu2}, D_{cpu2}^1, D_{gpu2}$, and D_{gpu2}^1 from the previous iteration. Note that before the λ -th iteration, D_{cpu2} and D_{cpu2}^1 need to be initialized to zero, and D_{gpu2} and D_{gpu2}^1 need to be initialized to $2^{\lambda-2}$. Then, we get D_{cpu1} and D_{gpu1} from the current iteration. After that, according to the following conditions to determine how to transfer data.

1. If $D_{gpu1} > D_{gpu2} + D_{gpu2}^1$, we need to transfer $D_{gpu1} - D_{gpu2} - D_{gpu2}^1$ elements from CPU to GPU, that is, copy each element of $A_{i-1}[D_{gpu2} + D_{gpu2}^1 + 1..D_{gpu1}]$ to $d_{-A_{i-1}}[D_{gpu2} + D_{gpu2}^1 + 1..D_{gpu1}]$; otherwise, we do not need to transfer data from CPU to GPU.
2. If $D_{cpu1} > D_{cpu2} + D_{cpu2}^1$, we need to transfer $D_{cpu1} - D_{cpu2} - D_{cpu2}^1$ elements from GPU to CPU, that is, copy each element of $d_{-A_{i-1}}[D_{gpu1} + 1..D_{gpu2} + D_{gpu2}^1]$ to $A_{i-1}[D_{gpu1} + 1..D_{gpu2} + D_{gpu2}^1]$; otherwise, we do not need to transfer data from GPU to CPU.

Algorithm 10 The improved generation stage using the incremental data transfer method

Require: $W_1 = [w_1, w_2, \dots, w_{n/2}]$, $d_{A_1} = [0, w_1]$, R_1, R_2

```

1: for  $i = 2$  to  $\lambda - 1$  do use the GPUs to generate the list  $d_{A_i}$ ; end for
2:  $D_{cpu2} = 0$ ,  $D_{cpu2}^1 = 0$ ,  $D_{gpu2} = 2^{\lambda-2}$ ,  $D_{gpu2}^1 = 2^{\lambda-2}$ ;
3: for  $i = \lambda$  to  $n/2$  do
4:    $\triangleright$  the add item process
5:    $D_1 = 2^{i-1}$ ,  $D_{cpu1} = \lfloor D_1 \times R_1 \rfloor$ ,  $D_{cpu1}^1 = \lfloor D_1 \times R_1 \rfloor$ ,  $D_{gpu1} = D_1 - D_{cpu1}$ ,  $D_{gpu1}^1 = D_1 - D_{cpu1}^1$ ;
6:    $k_{cpu1} = \min(c_{max}, D_{cpu1})$ ,  $k_{gpu1} = \min(g_{max}, D_{gpu1})$ ;
7:   #pragma omp parallel num_threads(2) {
8:     if  $omp\_tid = 0$  then
9:       if  $D_{gpu1} > D_{gpu2} + D_{gpu2}^1$ , then we copy each element of  $A_{i-1}[D_{gpu2} + D_{gpu2}^1 + 1..D_{gpu1}]$  to  $d_{A_{i-1}}[D_{gpu2} + D_{gpu2}^1 + 1..D_{gpu1}]$ ;
10:      use  $k_{gpu1}$  GPU threads to produce  $d_{A_{i-1}}^1[1..D_{gpu1}^1]$  by adding  $w_i$  to each element of  $d_{A_{i-1}}[1..D_{gpu1}]$ ;
11:     else
12:       if  $D_{cpu1} > D_{cpu2} + D_{cpu2}^1$ , then we copy each element of  $d_{A_{i-1}}[D_{gpu1} + 1..D_{gpu2} + D_{gpu2}^1]$  to  $A_{i-1}[D_{gpu1} + 1..D_{gpu2} + D_{gpu2}^1]$ ;
13:       use  $k_{cpu1}$  CPU threads to produce  $A_{i-1}^1[D_{gpu1}^1 + 1..D_1]$  by adding  $w_i$  to each element of  $A_{i-1}[D_{gpu1} + 1..D_1]$ ;
14:     end if }
15:    $\triangleright$  the partition and merge processes
16:    $D_2 = 2^{i-1}$ ,  $D_{cpu2} = \lfloor D_2 \times R_2 \rfloor$ ,  $D_{gpu2} = D_2 - D_{cpu2}$ ;
17:   perform Algorithm 2 to get  $D_{gpu2}^1$ ,  $D_{cpu2}^1 = D_2 - D_{gpu2}^1$ ;
18:    $k_{cpu2} = \min(c_{max}, 2^{\lfloor \log(D_{cpu2} + D_{cpu2}^1) \rfloor})$ ,  $k_{gpu2} = \min(g_{max}, 2^{\lfloor \log(D_{gpu2} + D_{gpu2}^1) \rfloor})$ ;
19:   #pragma omp parallel num_threads(2) {
20:     if  $omp\_tid = 0$  then
21:       if  $D_{gpu2} > D_{gpu1}$ , we copy each element of  $A_{i-1}[D_{gpu1} + 1..D_{gpu2}]$  to  $d_{A_{i-1}}[D_{gpu1} + 1..D_{gpu2}]$ ;
22:       if  $D_{gpu2}^1 > D_{gpu1}^1$ , we copy each element of  $A_{i-1}^1[D_{gpu1}^1 + 1..D_{gpu2}^1]$  to  $d_{A_{i-1}}^1[D_{gpu1}^1 + 1..D_{gpu2}^1]$ ;
23:       use  $k_{gpu2}$  GPU threads to merge  $d_{A_{i-1}}[1..D_{gpu2}]$  and  $d_{A_{i-1}}^1[1..D_{gpu2}^1]$  into  $d_{A_i}[1..D_{gpu2} + D_{gpu2}^1]$ ;
24:     else
25:       if  $D_{cpu2} > D_{cpu1}$ , we copy each element of  $d_{A_{i-1}}[D_{gpu2} + 1..D_{gpu1}]$  to  $A_{i-1}[D_{gpu2} + 1..D_{gpu1}]$ ;
26:       if  $D_{cpu2}^1 > D_{cpu1}^1$ , we copy each element of  $d_{A_{i-1}}^1[D_{gpu2}^1 + 1..D_{gpu1}^1]$  to  $A_{i-1}^1[D_{gpu2}^1 + 1..D_{gpu1}^1]$ ;
27:       use  $k_{cpu2}$  CPU threads to merge  $A_{i-1}[D_{gpu2} + 1..D_2]$  and  $A_{i-1}^1[D_{gpu2}^1 + 1..D_2]$  into  $A_i[D_{gpu2} + D_{gpu2}^1 + 1..2 \times D_2]$ ;
28:     end if }
29:   end for
30: return  $d_{A_{n/2}}[1..D_{gpu2} + D_{gpu2}^1]$  and  $A_{n/2}[D_{gpu2} + D_{gpu2}^1 + 1..2 \times D_2]$ 

```

Step 2: Before the partition and merge processes are executed, we first get D_{cpu2} , D_{cpu2}^1 , D_{gpu2} , and D_{gpu2}^1 from the current iteration and then according to the following conditions to determine how to transfer data.

1. If $D_{gpu2} > D_{gpu1}$, we need to transfer $D_{gpu2} - D_{gpu1}$ elements from CPU to GPU, that is, copy each element of $A_{i-1}[D_{gpu1} + 1..D_{gpu2}]$ to $d_{A_{i-1}}[D_{gpu1} + 1..D_{gpu2}]$.

2. If $D_{gpu2}^1 > D_{cpu1}^1$, we need to transfer $D_{gpu2}^1 - D_{cpu1}^1$ elements from CPU to GPU, that is, copy each element of $A_{i-1}^1[D_{cpu1}^1 + 1..D_{gpu2}^1]$ to $d_{-}A_{i-1}^1[D_{cpu1}^1 + 1..D_{gpu2}^1]$.
3. If $D_{cpu2} > D_{cpu1}$, we need to transfer $D_{cpu2} - D_{cpu1}$ elements from GPU to CPU, that is, copy each element of $d_{-}A_{i-1}[D_{gpu2} + 1..D_{cpu1}]$ to $A_{i-1}[D_{gpu2} + 1..D_{cpu1}]$.
4. If $D_{cpu2}^1 > D_{cpu1}^1$, we need to transfer $D_{cpu2}^1 - D_{cpu1}^1$ elements from GPU to CPU, that is, copy each element of $d_{-}A_{i-1}^1[D_{gpu2}^1 + 1..D_{cpu1}^1]$ to $A_{i-1}^1[D_{gpu2}^1 + 1..D_{cpu1}^1]$.

Secondly, we describe the use of the incremental data transfer method during the pruning and search stages. The use of the method includes the following two cases:

Case 1: We get D_{gpu2} and D_{gpu2}^1 from the last iteration of generating the list $d_{-}A$. If $D_{gpu3} > D_{gpu2} + D_{gpu2}^1$, we need to transfer $D_{gpu3} - D_{gpu2} - D_{gpu2}^1$ elements from CPU to GPU, that is, copy each element of $A[D_{gpu2} + D_{gpu2}^1 + 1..D_{gpu3}]$ to $d_{-}A[D_{gpu2} + D_{gpu2}^1 + 1..D_{gpu3}]$. If $D_{gpu3} < D_{gpu2} + D_{gpu2}^1$, we need to transfer $D_{gpu2} + D_{gpu2}^1 - D_{gpu3}$ elements from GPU to CPU, that is, copy each element of $d_{-}A[D_{gpu3} + 1..D_{gpu2} + D_{gpu2}^1]$ to $A[D_{gpu3} + 1..D_{gpu2} + D_{gpu2}^1]$.

Case 2: We get D_{gpu2} and D_{gpu2}^1 from the last iteration of generating the list $d_{-}B$. We copy each element of $B[D_{gpu2} + D_{gpu2}^1 + 1..D_3]$ to $d_{-}B[D_{gpu2} + D_{gpu2}^1 + 1..D_3]$, and copy each element of $d_{-}B[1..D_{gpu2} + D_{gpu2}^1]$ to $B[1..D_{gpu2} + D_{gpu2}^1]$.

As described earlier, it is not hard to see that the incremental data transfer method can help us to avoid transferring large amounts of repeated data back and forth between CPUs and GPUs, which greatly reduces the CPU-GPU communication overhead.

4.2. The feedback-based dynamic task distribution scheme

We can statically assign tasks to both CPUs and GPUs by using our proposed task distribution model. However, load imbalance between CPUs and GPUs may arise during runtime; this may cause a significant decrease in performance. To balance the workload between CPUs and GPUs during runtime, a feedback-based dynamic task distribution scheme is designed in this section.

To facilitate our discussion, let us suppose that a large task can be broken down into n separate subtasks $T = [T_1, T_2, \dots, T_n]$. The task T needs to execute n iterations to complete, and we use both CPUs and GPUs to cooperatively execute the subtask T_i during the i -th iteration, where $1 \leq i \leq n$. The total workload WL of the task T is divided into several parts and each part is assigned to both CPUs and GPUs. Let WL_i denote the workload of the subtask T_i , let $WL_{cpu.i}$ denote the workload assigned to the CPUs, and let $WL_{gpu.i}$ denote the workload assigned to the GPUs, where $WL = \sum_{i=1}^n WL_i$, $WL_i = WL_{cpu.i} + WL_{gpu.i}$, and $1 \leq i \leq n$. For clarity, we use the task execution speed as load index, which is a measure of load status. Let $T_{cpu.i}$ and $V_{cpu.i}$ denote the execution time and the execution speed of the subtask T_i on the host side, respectively, where $1 \leq i \leq n$. We obtain the following equation:

$$V_{cpu.i} = WL_{cpu.i} / T_{cpu.i}. \quad (12)$$

Similarly, let $T_{gpu.i}$ and $V_{gpu.i}$ denote the execution time and the execution speed of the subtask T_i on the device side, respectively, where $1 \leq i \leq n$. We obtain the following equation:

$$V_{gpu.i} = WL_{gpu.i} / T_{gpu.i}. \quad (13)$$

The feedback-based dynamic task distribution scheme is shown in Figure 2, which consists of the following three steps:

Step 1: Before the subtask T_i is executed in the i -th iteration, we assign the workload WL_i of the subtask T_i to both CPUs and GPUs according to the task distribution ratio TDR_i , where $1 \leq i \leq n$. The workload assigned to the CPUs and GPUs can be calculated as $WL_{cpu.i} = \lfloor WL_i \times TDR_i \rfloor$, and $WL_{gpu.i} = WL_i - WL_{cpu.i}$. Note that the initial task distribution ratio TDR_1 can be obtained from Equation (10), or can be specified as 50.0%.

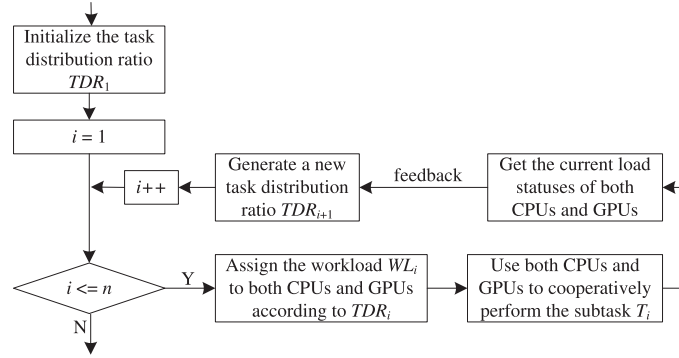


Figure 2. The feedback-based dynamic task distribution scheme.

Step 2: After the subtask T_i has been completed in the i -th iteration, we get the current load statuses of both CPUs and GPUs, where $1 \leq i \leq n - 1$. Specifically, we first achieve the task execution times $T_{cpu,i}$ and $T_{gpu,i}$. Then, we calculate the task execution speeds $V_{cpu,i}$ and $V_{gpu,i}$ by using Equations (12) and (13), respectively. Finally, according to the current load statuses, we generate a new task distribution ratio TDR_{i+1} , which will be used in the next iteration and can be calculated as follows:

$$TDR_{i+1} = \frac{V_{cpu,i}}{V_{cpu,i} + V_{gpu,i}}. \quad (14)$$

Step 3: Repeat Steps 1–2 until all n subtasks have been executed.

The following presentations describe how to apply the feedback-based dynamic task distribution scheme to the three stages of the parallel *two-list* algorithm.

In the case of the generation stage, it is clear that the whole generation procedure needs to execute $n/2 - 1$ iterations to complete, and we only need to consider the load balancing between CPUs and GPUs when $\lambda \leq i \leq n/2$. For simplicity, let us take the add item process as an example to illustrate how to use the dynamic task distribution scheme, as shown in Algorithm 11. Note that when $i = \lambda$,

Algorithm 11 The improved add item process using the dynamic task distribution scheme

Require: $W_1 = [w_1, w_2, \dots, w_{n/2}]$, $d_{A_1} = [0, w_1]$, R_1, R_2

```

1: for i = λ to n/2 do
2:   if i = λ, then TDR_i = R_1;
3:   WL_i = 2^{i-1}, WL_{cpu,i} = ⌊WL_i × TDR_i⌋, WL_{gpu,i} = WL_i - WL_{cpu,i};
4:   D_{cpu1} = WL_{cpu,i}, D_{cpu1}^1 = WL_{cpu,i}, D_{gpu1} = WL_{gpu,i}, D_{gpu1}^1 = WL_{gpu,i};
5:   k_{cpu1} = min(c_{max}, D_{cpu1}), k_{gpu1} = min(g_{max}, D_{gpu1});
6:   #pragma omp parallel num_threads(2) {
7:     if omp_tid = 0 then
8:       getStartTime(T_{gpu_start});
9:       use k_{gpu1} GPU threads to produce d_{A_{i-1}}^1[1..D_{gpu1}^1] by adding w_i to each element of
10:      d_{A_{i-1}}[1..D_{gpu1}];
11:       getFinishTime(T_{gpu_finish});
12:     else
13:       getStartTime(T_{cpu_start});
14:       use k_{cpu1} CPU threads to produce A_{i-1}^1[D_{gpu1}^1 + 1..D_1] by adding w_i to each element
15:       of A_{i-1}[D_{gpu1} + 1..D_1];
16:       getFinishTime(T_{cpu_finish});
17:     end if }
18:   T_{cpu,i} = T_{cpu_finish} - T_{cpu_start}, T_{gpu,i} = T_{gpu_finish} - T_{gpu_start};
19:   obtain V_{cpu,i}, V_{gpu,i} and TDR_{i+1} by using Equations (12), (13) and (14) respectively;
20: end for
  
```

R_1 is used as the initial task distribution ratio; when $\lambda < i \leq n/2$, we dynamically generate a new task distribution ratio according to the current load statuses of both CPUs and GPUs.

In the case of the pruning and search stages, in order to better apply our dynamic task distribution scheme, we first divide d_A into v equal blocks, where each block contains $2^{n/2}/v$ elements. Let $d_A = [d_A_1, d_A_2, \dots, d_A_i, \dots, d_A_v]$, where $d_A_i = [a_{i,1}, a_{i,2}, \dots, a_{i,2^{n/2}/v}]$ and $1 \leq i \leq v$. Then, we execute v iterations to complete the whole pruning and search stages. We use both CPUs and GPUs to cooperatively perform the pruning and search operations for the block d_A_i and the list d_B in the i -th iteration, the total workload WL_i is equal to $2^{n/2}/v$, where $1 \leq i \leq v$.

5. EXPERIMENTAL EVALUATION

In this section, we first present the experimental setup, next, validate the proposed task distribution model, then analyze the effectiveness of the proposed performance optimization schemes, and finally, evaluate the performance of the proposed CPU-GPU cooperative implementation.

5.1. Experimental setup

In our experiments, we use three different methods to implement the parallel *two-list* algorithm as follows: CPU-only implementation, namely, we implement it on the CPUs using OpenMP; GPU-only implementation, namely, we implement it on the GPUs using CUDA; CPU-GPU cooperative implementation, namely, we implement it on both CPUs and GPUs using OpenMP and CUDA. We compare their performance with Horowitz and Sahni's *two-list* algorithm [2], which is the best known sequential algorithm for solving SSP. The experiments are carried out on the following three different test platforms:

- Test platform 1: Dual Intel Xeon E5504 CPUs (4 cores at 2.0 GHz) (Intel Corporation, Santa Clara, CA, USA), 32GB of main memory, and a GTX 465 GPU (352 CUDA cores at 607 MHz, 1 GB memory, 102.6 GB/s memory bandwidth) (NVIDIA Corporation, Santa Clara, CA, USA).
- Test platform 2: Dual Intel Xeon E5-2620 CPUs (6 cores at 2.0 GHz), 32 GB of main memory, and a Tesla M2090 GPU (512 CUDA cores at 1.3 GHz, 6 GB memory, 177.6 GB/s memory bandwidth).
- Test platform 3: Dual Intel Xeon E5-2650 CPUs (8 cores at 2.0 GHz), 32 GB of main memory, and a Tesla K20m GPU (2496 CUDA cores at 706 MHz, 5 GB memory, 208 GB/s memory bandwidth).

In software, the testing platform is built on top of the SUSE Linux Enterprise 11 operating system (SUSE Linux Company, Nurnberg, Germany) with NVIDIA CUDA driver version 5.5 and GCC version 4.4.7.

Considering Horowitz and Sahni's sequential *two-list* algorithm [2] and the parallel *two-list* algorithm of Li *et al.* [6] all require $O(2^{n/2})$ memory space, implying that the problem size is limited by the available memory. Therefore, we test the following seven different problem sizes: 42, 44, 46, 48, 50, 52, and 54. For each problem size, we use a random number generator to produce 100 different instances of SSP. The average execution time of 100 different instances is considered and it is measured in milliseconds. Each instance presents the following features: (1) w_i is randomly drawn in $[1, 10^8]$, $i \in \{1, \dots, n\}$; (2) $M = \alpha \sum_{i=1}^n w_i$, $\alpha \in \{0.2, \dots, 0.8\}$; and (3) $w_i < M$, $i \in \{1, \dots, n\}$.

5.2. Evaluation of the task distribution model

According to our proposed task distribution model, the calculation of the task distribution ratio depends on T_{cpu} , T_{gpu} , and T_{comm} . In order to estimate the task distribution ratio R_1 of the add item process, for each instance, we specify $M = 0.5 \sum_{i=1}^n w_i$ and conduct our experiments as follows. Firstly, we run the add item process on the CPUs to obtain T_{cpu} . Next, we run it on a GPU to obtain T_{gpu} . Then, we specify $R_1 = 1$ and run it on both CPUs and GPUs to obtain T_{comm} . Finally, we calculate R_1 by using Equation (10). Similarly, we can obtain the task distribution ratios R_2 and R_3 .

Table IV. The estimated task distribution ratio of the add item process for different problem sizes.

n	Test platform 1				Test platform 2				Test platform 3			
	T_{cpu}	T_{gpu}	T_{comm}	$R_1(\%)$	T_{cpu}	T_{gpu}	T_{comm}	$R_1(\%)$	T_{cpu}	T_{gpu}	T_{comm}	$R_1(\%)$
42	11.52	3.81	7.60	16.62	7.70	3.10	6.83	17.58	6.26	2.46	6.05	16.66
44	18.79	5.98	12.30	16.13	12.53	4.81	10.16	17.49	10.19	3.96	9.76	16.56
46	31.88	9.80	22.25	15.33	21.28	8.17	18.38	17.08	17.31	6.71	17.29	16.24
48	56.66	17.09	42.70	14.68	37.81	14.24	35.28	16.31	30.74	11.67	32.99	15.48
50	106.08	31.49	85.19	14.14	70.75	26.20	70.41	15.65	57.28	21.41	65.36	14.86
52	202.64	59.81	161.49	14.11	135.59	49.90	135.08	15.57	109.94	40.96	126.30	14.78
54	407.85	120.08	323.55	14.10	273.25	100.12	270.21	15.56	220.35	81.84	252.65	14.75

Table V. The estimated task distribution ratio of the partition and merge processes for different problem sizes.

n	Test platform 1				Test platform 2				Test platform 3			
	T_{cpu}	T_{gpu}	T_{comm}	$R_2(\%)$	T_{cpu}	T_{gpu}	T_{comm}	$R_2(\%)$	T_{cpu}	T_{gpu}	T_{comm}	$R_2(\%)$
42	92.44	67.13	88.21	27.09	70.21	56.58	73.51	28.25	63.41	46.45	68.51	26.04
44	150.80	102.53	138.73	26.15	114.25	85.43	115.61	27.10	103.32	70.82	107.75	25.12
46	255.84	168.02	231.60	25.63	194.10	140.06	193.00	26.57	175.49	115.13	179.88	24.47
48	454.75	293.08	419.84	25.10	344.90	244.19	349.87	26.01	311.56	200.11	326.08	23.89
50	851.32	539.95	797.42	24.67	645.34	449.38	664.52	25.54	580.62	367.16	619.33	23.43
52	1626.25	1025.63	1509.05	24.65	1236.79	854.76	1257.54	25.52	1114.29	702.37	1184.03	23.41
54	3273.09	2059.20	3027.46	24.63	2492.43	1716.95	2522.88	25.50	2233.41	1403.38	2363.32	23.39

Table VI. The estimated task distribution ratio of the pruning and search stages for different problem sizes.

n	Test platform 1				Test platform 2				Test platform 3			
	T_{cpu}	T_{gpu}	T_{comm}	$R_3(\%)$	T_{cpu}	T_{gpu}	T_{comm}	$R_3(\%)$	T_{cpu}	T_{gpu}	T_{comm}	$R_3(\%)$
42	17.96	6.32	9.38	18.78	13.46	5.25	8.44	19.34	12.03	4.36	7.97	17.90
44	29.06	9.99	18.26	17.43	21.73	8.36	15.05	18.52	19.45	6.88	14.13	17.00
46	48.72	15.84	33.27	16.19	36.47	13.20	24.79	17.73	32.65	10.85	23.43	16.21
48	85.34	27.15	58.66	15.86	63.86	22.62	49.51	16.63	57.12	18.54	46.14	15.22
50	156.76	49.49	110.06	15.65	117.25	41.19	98.60	16.02	104.45	33.65	91.90	14.63
52	293.53	92.21	207.28	15.55	220.26	76.94	185.44	15.94	196.48	63.15	174.53	14.55
54	575.96	180.52	409.97	15.48	432.75	150.52	369.70	15.79	383.94	123.03	342.56	14.48

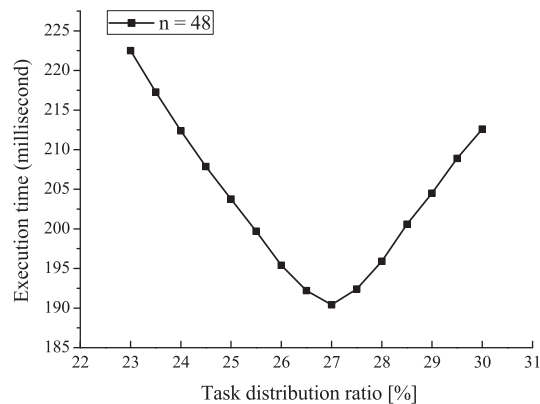


Figure 3. The execution time of the partition and merge processes for different task distribution ratios.

Tables IV, V, and VI show the estimated task distribution ratios of the add item process, the partition and merge processes, and the pruning and search stages, respectively, for the seven different problem sizes on three different test platforms.

In order to verify whether the estimated task distribution ratio is reasonable, we specify $n = 48$ and test the execution time of the partition and merge processes by using 15 different distribution ratios on Test Platform 2. The task distribution ratio is varied from 23.0% to 30.0% at 0.5% intervals. The results are shown in Figure 3. We can see that the actual task distribution ratio for minimal execution time is approximately 27%, whereas the corresponding estimated task distribution ratio is 26.01%. It is clear that the estimated task distribution ratio has only 1% error. Based on our investigation and analysis, we find that the causes of producing error results are mainly associated with the CPU cache occupancy and the GPU processor occupancy. The execution time with the estimated task distribution ratio is 195.41 milliseconds, while with the actual optimal task distribution ratio is 190.41 milliseconds. Hence, the error is acceptable.

We conduct similar experiments for all the other instances. Table VII shows the actual optimal task distribution ratios R_1 , R_2 , and R_3 for different problem sizes on three different test platforms. According to Tables IV–VII, it is not hard to see that the estimated task distribution ratios are close to the actual optimal values for different problem sizes. The actual optimal values are 0.78–1.06% higher than the estimated values. The results show that our proposed task distribution model can find reasonable task distribution ratio.

5.3. Analysis of the CPU-GPU communication optimization

To evaluate the effectiveness of the proposed CPU-GPU communication optimization scheme, for each instance, we specify $M = 0.5 \sum_{i=1}^n w_i$ and conduct our experiments as follows. Firstly, we run the parallel *two-list* algorithm on both CPUs and GPUs by using the actual optimal task distribution ratio presented in Table VII. Secondly, the incremental data transfer method is adopted, and we run the algorithm again on both CPUs and GPUs. Finally, the performance of the CPU-GPU cooperative implementation without communication optimization (CGCI without CO) is compared with that of the CPU-GPU cooperative implementation with communication optimization (CGCI with CO).

Table VII. The actual optimal task distribution ratio.

n	Test platform 1			Test platform 2			Test platform 3		
	$R_1(\%)$	$R_2(\%)$	$R_3(\%)$	$R_1(\%)$	$R_2(\%)$	$R_3(\%)$	$R_1(\%)$	$R_2(\%)$	$R_3(\%)$
42	17.53	28.12	19.81	18.55	29.28	20.40	17.58	26.99	18.88
44	17.02	27.17	18.39	18.45	28.13	19.54	17.47	26.08	17.94
46	16.17	26.64	17.08	18.02	27.59	18.71	17.13	25.41	17.10
48	15.49	26.10	16.73	17.21	27.02	17.54	16.33	24.82	16.06
50	14.92	25.66	16.51	16.51	26.54	16.90	15.68	24.34	15.43
52	14.89	25.64	16.41	16.43	26.52	16.82	15.59	24.32	15.35
54	14.88	25.62	16.33	16.42	26.49	16.66	15.56	24.30	15.28

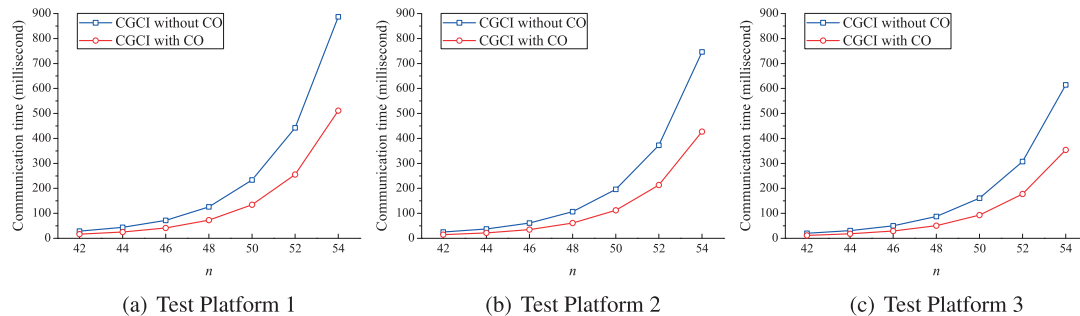


Figure 4. The CPU-GPU communication time comparison between CGCI without CO and CGCI with CO. (a) Test Platform 1; (b) Test Platform 2; and (c) Test Platform 3.

Figure 4 shows the CPU-GPU communication time comparison between CGCI without CO and CGCI with CO on three different test platforms. In Figure 4, the line marked as ‘CGCI without CO’ represents the CPU-GPU communication time of the CPU-GPU cooperative implementation without communication optimization, and the line marked as ‘CGCI with CO’ denotes the CPU-GPU communication time of the CPU-GPU cooperative implementation with communication optimization. Compared with the CGCI with CO, the communication times of the CGCI without CO are reduced by an average of 42.15%, 42.58%, and 41.94% on Test Platform 1, Test Platform 2, and Test Platform 3, respectively. It is obvious that the CPU-GPU communication time can be greatly reduced after the incremental data transfer method has been used.

Table VIII shows the performance benefits achieved through using the incremental data transfer method on three different test platforms. In Table VIII, the column marked as ‘CGCI without CO’ represents the execution time of the CPU-GPU cooperative implementation without communication optimization, and the column marked as ‘CGCI with CO’ denotes the execution time of the CPU-GPU cooperative implementation with communication optimization. The performance comparison between CGCI without CO and CGCI with CO shows that our communication optimization scheme significantly improves the performance of the CPU-GPU cooperative implementation. Compared with the CGCI without CO, the CGCI with CO achieves an average of 6.36%, 7.21%, and 7.00% performance improvements on Test Platform 1, Test Platform 2, and Test Platform 3, respectively.

5.4. Analysis of the dynamic load balancing scheme

To evaluate the effectiveness of the proposed feedback-based dynamic task distribution scheme, we run the parallel *two-list* algorithm again on both CPUs and GPUs to obtain the execution time of the CPU-GPU cooperative implementation with both communication optimization and dynamic load balancing (CGCI with CO & DLB).

Table VIII. Benefits of the CPU-GPU communication optimization on three different test platforms.

<i>n</i>	Test platform 1			Test platform 2			Test platform 3		
	CGCI without CO	CGCI with CO	Benefit (%)	CGCI without CO	CGCI with CO	Benefit (%)	CGCI without CO	CGCI with CO	Benefit (%)
42	61.47	57.77	6.40	52.92	49.33	7.28	43.87	40.97	7.08
44	94.21	88.55	6.39	78.43	73.12	7.26	67.55	63.10	7.05
46	154.40	145.14	6.38	128.24	119.58	7.24	110.26	103.02	7.03
48	270.94	254.72	6.37	224.11	209.01	7.22	192.13	179.54	7.01
50	501.78	471.80	6.35	413.42	385.67	7.20	353.45	330.36	6.99
52	951.88	895.41	6.31	786.70	734.28	7.14	675.68	631.89	6.93
54	1907.64	1794.58	6.30	1577.32	1472.48	7.12	1349.34	1262.13	6.91

CGCI without CO, CPU-GPU cooperative implementation without communication optimization; CGCI with CO, CPU-GPU cooperative implementation with communication optimization.

Table IX. Benefits of the feedback-based dynamic task distribution scheme on three different test platforms.

<i>n</i>	Test platform 1			Test platform 2			Test platform 3		
	CGCI with CO	CGCI with CO & DLB	Benefit (%)	CGCI with CO	CGCI with CO & DLB	Benefit (%)	CGCI with CO	CGCI with CO & DLB	Benefit (%)
42	57.77	56.09	2.98	49.33	47.93	2.93	40.97	39.74	3.09
44	88.55	85.99	2.99	73.12	71.03	2.95	63.10	61.20	3.11
46	145.14	140.91	3.00	119.58	116.14	2.96	103.02	99.87	3.12
48	254.72	247.26	3.02	209.01	202.97	2.97	179.54	174.09	3.13
50	471.80	457.92	3.03	385.67	374.52	2.98	330.36	320.30	3.14
52	895.41	868.98	3.04	734.28	712.97	2.99	631.89	612.59	3.15
54	1794.58	1741.47	3.05	1472.48	1429.58	3.00	1262.13	1223.47	3.16

CGCI with CO, CPU-GPU cooperative implementation with communication optimization; CGCI with CO & DLB, CPU-GPU cooperative implementation with both communication optimization and dynamic load balancing.

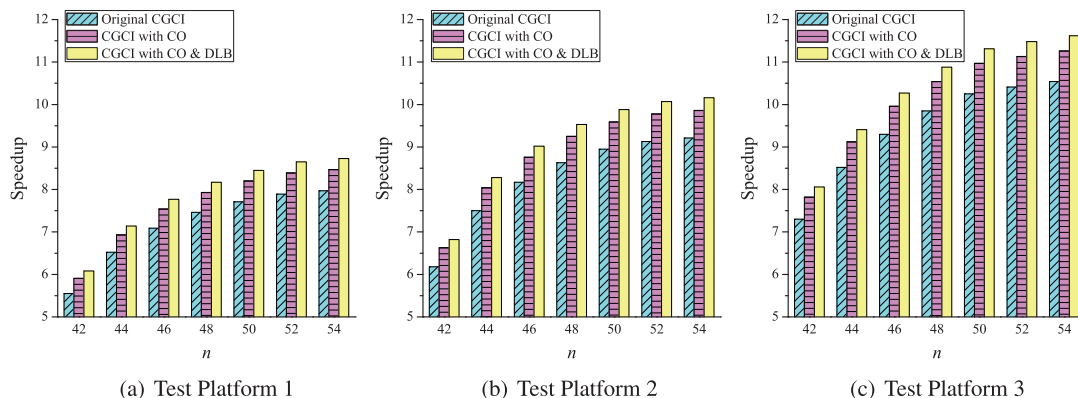


Figure 5. The performance comparison among the original CGCI, CGCI with CO, and CGCI with CO and & DLB. (a) Test Platform 1; (b) Test Platform 2; and (c) Test Platform 3.

Table IX shows the performance benefits achieved through using the feedback-based dynamic task distribution scheme on three different test platforms. The results show that the proposed dynamic task distribution scheme also effectively improves the performance of the cooperative implementation, and the load balancing result is stable across different problem sizes. Compared with the CGCI with CO, the CGCI with CO & DLB achieves an average of 3.02%, 2.97%, and 3.13% performance improvements on Test Platform 1, Test Platform 2, and Test Platform 3, respectively.

From Tables VIII and IX, it is clear that our proposed performance optimization schemes give a significant performance benefit. Figure 5 shows the performance comparison among the original CGCI, CGCI with CO, and CGCI with CO & DLB on three different test platforms. Compared with the original CGCI, the CGCI with CO & DLB achieves an average of 9.57%, 10.39%, and 10.35% performance improvements on Test Platform 1, Test Platform 2, and Test Platform 3, respectively.

5.5. Performance evaluation of the CPU-GPU cooperative implementation

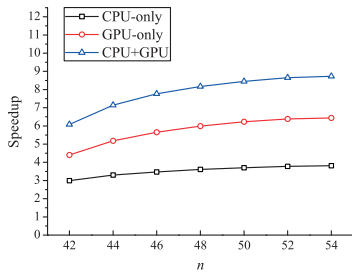
In this section, we first compare the performance of the CPU-GPU cooperative implementation with that of the CPU-only and GPU-only implementations, then analyze the performance of the CPU-GPU cooperative implementation under three different test platforms, and finally, evaluate the performance of the CPU-GPU cooperative implementation with three different knapsack capacities.

5.5.1. Comparison with CPU-only / GPU-only implementation. In order to accurately evaluate the performance of the proposed CPU-GPU cooperative implementation, for each instance, we specify $M = 0.5 \sum_{i=1}^n w_i$ and conduct a series of experiments as follows. Firstly, we run Horowitz and Sahni's sequential *two-list* algorithm on a single CPU. Secondly, we run the parallel *two-list* algorithm on two multi-core CPUs. Thirdly, we run the parallel algorithm on a single GPU. Fourthly, we run the parallel algorithm on both CPUs and GPUs. Finally, the speedups are calculated for the CPU-only, GPU-only, and CPU-GPU cooperative implementations, respectively. Note that the speedup is defined as the sequential execution time over the parallel execution time.

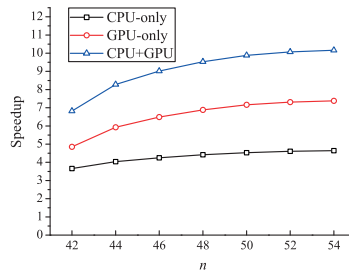
Table X shows the execution time of the sequential implementation, CPU-only implementation, GPU-only implementation, and CPU-GPU cooperative implementation on three different test platforms for the seven different problem sizes. Figure 6(a)–(c) shows the speedup comparison among the CPU-only, GPU-only, and CPU-GPU cooperative implementations on three different test platforms. Obviously, the CPU-only, GPU-only, and CPU-GPU cooperative implementations has much better performance than the sequential implementation. For instance, under Test Platform 3, the speedup of the CPU-only implementation increases from 3.98 \times to 5.09 \times , the speedup of the GPU-only implementation increases from 5.99 \times to 8.84 \times , and the speedup of the CPU-GPU cooperative implementation increases from 8.06 \times to 11.62 \times , when the problem size scales from 42 to 54. The results also show that when the problem size increases, the speedups of the CPU-only,

Table X. The execution time of four different implementations on three different test platforms.

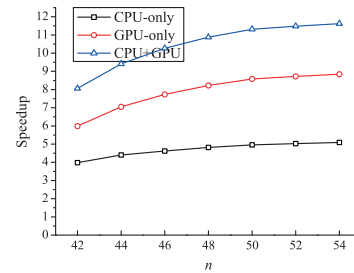
n	Test platform 1				Test platform 2				Test platform 3			
	Sequential	CPU-only	GPU-only	CPU+GPU	Sequential	CPU-only	GPU-only	CPU+GPU	Sequential	CPU-only	GPU-only	CPU+GPU
42	341.27	114.30	77.56	56.09	327.10	89.46	67.44	47.93	320.28	80.42	53.47	39.74
44	613.83	186.23	118.50	85.99	588.05	145.40	99.17	71.03	575.78	130.89	81.67	61.20
46	1094.18	315.40	193.66	140.91	1047.68	246.60	161.43	116.14	1025.80	221.93	132.70	99.87
48	2020.53	559.45	337.32	247.26	1933.65	437.26	281.05	202.97	1893.24	393.18	230.32	174.09
50	3868.32	1044.53	620.92	457.92	3700.05	815.96	516.77	374.52	3622.68	730.75	422.22	320.30
52	7513.38	1989.77	1177.65	868.98	7182.79	1559.43	982.60	712.97	7032.53	1398.52	806.48	612.59
54	15197.11	3990.83	2359.80	1741.47	14520.83	3131.73	1967.59	1429.58	14216.89	2793.38	1608.25	1223.47



(a) Test Platform 1

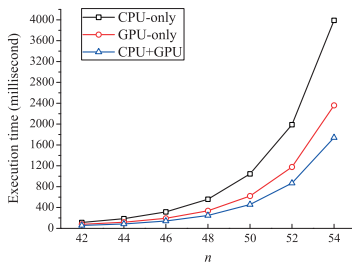


(b) Test Platform 2

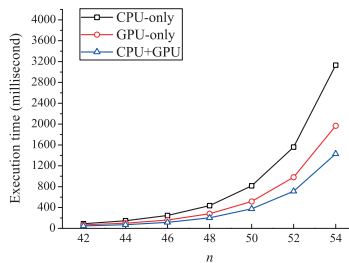


(c) Test Platform 3

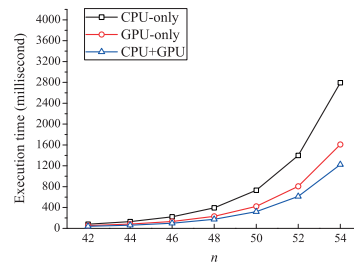
Figure 6. The speedup comparison among the CPU-only, GPU-only and CPU-GPU cooperative implementations. (a) Test Platform 1; (b) Test Platform 2; and (c) Test Platform 3.



(a) Test Platform 1



(b) Test Platform 2



(c) Test Platform 3

Figure 7. The execution time comparison among the CPU-only, GPU-only and CPU-GPU cooperative implementations. (a) Test Platform 1; (b) Test Platform 2; and (c) Test Platform 3.

GPU-only, and CPU-GPU cooperative implementations grow accordingly. Note that the larger the problem size is, the slower the speedup increases, and the speedup will gradually reach a peak. In the case of the CPU-GPU cooperative implementation, its speedup is not substantial for small problem sizes, this is mainly because there is not enough work to fully utilize the computational power of both CPUs and GPUs. However, it can achieve substantial speedup for large problem sizes. For example, when $n = 54$, it obtains speedup of 8.73, 10.16 and 11.62 \times over the best sequential implementation on Test Platform 1, Test Platform 2, and Test Platform 3, respectively. Therefore, our proposed CPU-GPU cooperative implementation is suitable for large-scale SSP.

Figure 7(a)–(c) shows the execution time comparison among the CPU-only, GPU-only, and CPU-GPU cooperative implementations on three different test platforms. Compared with the CPU-only implementation, the execution times of the cooperative implementation on Test Platform 1, Test Platform 2, and Test Platform 3 are reduced by an average of 56.00%, 53.85%, and 55.86%, respectively. Compared with the GPU-only implementation, the execution times of the cooperative implementation on Test Platform 1, Test Platform 2, and Test Platform 3 are reduced by an average of 26.82%, 27.93%, and 24.57%, respectively. The results show that the CPU-GPU cooperative

implementation significantly outperforms the CPU-only and GPU-only cases. We believe that this is because all the computational power of both CPUs and GPUs has been fully utilized.

5.5.2. *Performance evaluation under different test platforms.* Figure 8 illustrates the speedups of the CPU-GPU cooperative implementation obtained on three different test platforms. Apparently, Test Platform 3 yields better performance than the other two test platforms. Test Platform 3 achieves an average of 32.77% and 14.67% performance improvements over Test Platform 1 and Test Platform 2, respectively, indicating that our approach has good scalability. We believe that the more computing power of a heterogeneous CPU-GPU system is utilized, the better is the performance.

5.5.3. *Performance evaluation with different knapsack capacities.* Considering the real-world applications and the worst-case execution time, it is necessary to take into account different knapsack capacities. In our experiments, we specify three different capacity values: $M = \alpha \sum_{i=1}^n w_i$, $\alpha \in \{0.3, 0.5, 0.8\}$. Note that we may need to adopt different task distribution ratios for different knapsack capacities, therefore we conduct the experiments as described in Section 5.2 to obtain the optimal task distribution ratios for $M = 0.3 \sum_{i=1}^n w_i$ and $M = 0.8 \sum_{i=1}^n w_i$, respectively.

Figure 9 shows the speedups of the CPU-GPU cooperative implementation with three different knapsack capacities obtained on three different test platforms. Clearly, the CPU-GPU cooperative implementation achieves a higher speedup when $M = 0.5 \sum_{i=1}^n w_i$ and it achieves a lower speedup when $M = 0.8 \sum_{i=1}^n w_i$. The results reveal that the knapsack capacity could affect the running time. Specifically, when the capacity value is $0.3 \sum_{i=1}^n w_i$, in comparison with $0.5 \sum_{i=1}^n w_i$, the total execution times are increased by an average of 3.87%, 3.97%, and 4.03% on

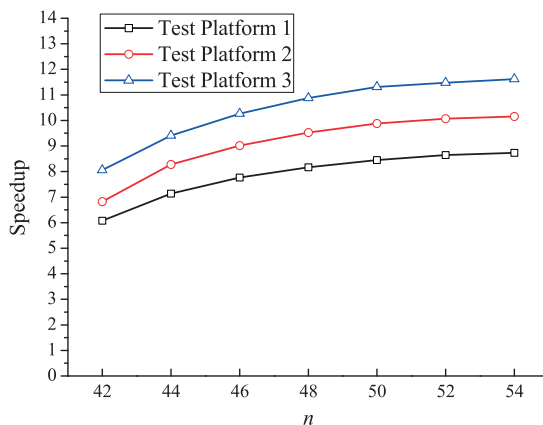


Figure 8. The speedups of the CPU-GPU cooperative implementation obtained on three different test platforms.

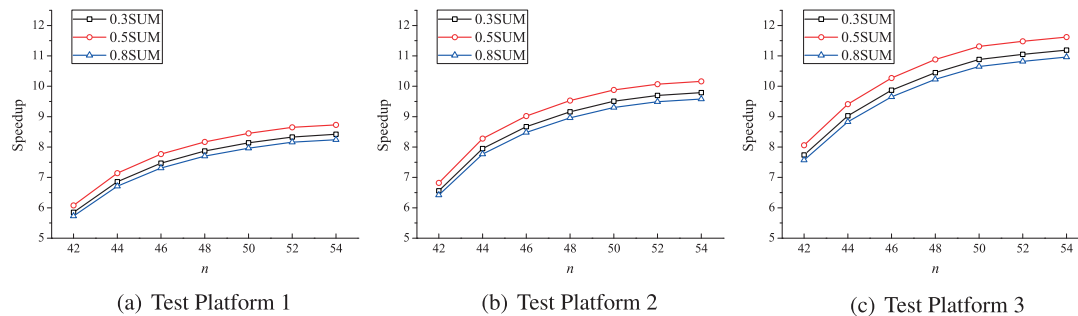


Figure 9. The speedups of the CPU-GPU cooperative implementation with three different knapsack capacities. (a) Test Platform 1; (b) Test Platform 2; and (c) Test Platform 3.

Test Platform 1, Test Platform 2, and Test Platform 3, respectively. Similarly, when the capacity value is $0.8 \sum_{i=1}^n w_i$, in comparison with $0.5 \sum_{i=1}^n w_i$, the total execution times are increased by an average of 6.15%, 6.27%, and 6.30% on Test Platform 1, Test Platform 2, and Test Platform 3, respectively. In fact, whether the knapsack capacity is small or large, this does not affect the execution time of the generation stage, but this can affect the execution times of both the pruning stage and the search stage.

6. CONCLUSIONS

In this paper, an original CPU-GPU cooperative implementation of the parallel *two-list* algorithm is proposed to efficiently solve SSP. In order to find the most appropriate task distribution ratio between CPUs and GPUs, a simple but effective task distribution model is established, and the experimental results prove that the proposed model can find reasonable task distribution ratio. To improve the performance of the CPU-GPU cooperative implementation, an incremental data transfer method is proposed to reduce the CPU-GPU communication overhead, a feedback-based dynamic task distribution scheme is designed to effectively balance the workload between CPUs and GPUs during runtime, and the experimental results demonstrate that the proposed performance optimization schemes can greatly improve the computational efficiency. A series of experiments are conducted to compare the performance of the CPU-GPU cooperative implementation with that of the best sequential implementation, the CPU-only implementation, and the GPU-only implementation. The experimental results show that the CPU-GPU cooperative implementation yields significant performance benefits by fully utilizing all the computational power of both CPUs and GPUs. Specifically, the CPU-GPU cooperative implementation produces a speedup factor of $11.62\times$ over the best sequential implementation and achieves up to an average of 123.51% and 36.00% performance improvements over the CPU-only case and the GPU-only case, respectively.

Although we only explore the CPU-GPU cooperative computing technology in a heterogeneous system with two CPUs and one GPU, the approach can be easily employed to a heterogeneous system with multiple CPUs and multiple accelerators (such as GPUs and/or many integrated cores (MICs)). In future work, we will develop a high-level directive-based heterogeneous cooperative parallel programming model, which allows programmers to deal with the tedious and complex heterogeneous cooperative computing in a simpler manner and directly to the sequential code. We expect that the programming model can automatically and reasonably map computations and data across CPUs and accelerators.

ACKNOWLEDGEMENTS

The authors would like to thank the editor and anonymous reviewers for their valuable suggestions. This research was partially funded by the Key Program of National Natural Science Foundation of China (Grant Nos. 61133005, 61432005), the National Natural Science Foundation of China (Grant Nos. 61370095, 61472124, 61472126), and the International Science & Technology Cooperation Program of China (Grant No. 2015DFA11240).

REFERENCES

1. Martello S, Toth P. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Inc.: New York, NY, 1990.
2. Horowitz E, Sahni S. Computing Partitions with Applications to the Knapsack Problem. *Journal of the ACM (JACM)* 1974; **21**(2):277–292.
3. Karnin ED. A parallel algorithm for the knapsack problem. *IEEE Transactions on Computers* 1984; **100**(5):404–408.
4. Lou DC, Chang CC. A parallel two-list algorithm for the knapsack problem. *Parallel Computing* 1997; **22**(14):1985–1996.
5. Sanches CAA, Soma NY, Yanasse HH. An optimal and scalable parallelization of the *two-list* algorithm for the subset-sum problem. *European Journal of Operational Research* 2007; **176**(2):870–879.
6. Li KL, Li RF, Li QH. Optimal parallel algorithms for the knapsack problem without memory conflicts. *Journal of Computer Science and Technology* 2004; **19**(6):760–768.

7. Che S, Boyer M, Meng JY, Tarjan D, Sheaffer JW, Skadron K. A performance study of general-purpose applications on graphics processors using CUDA. *Journal of Parallel and Distributed Computing* 2008; **68**(10):1370–1380.
8. Huang QH, Huang ZY, Werstein P, Purvis M. GPU as a general purpose computing resource. *Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies, 2008. PDCAT 2008*, IEEE, 2008; 151–158.
9. Bokhari SS. Parallel solution of the subset-sum problem: an empirical study. *Concurrency and Computation: Practice and Experience* 2012; **24**(18):2241–2254.
10. Wan LJ, Li KL, Liu J, Li KQ. GPU implementation of a parallel two-list algorithm for the subset-sum problem. *Concurrency and Computation: Practice and Experience* 2015; **27**(1):119–145.
11. Boyer V, El Baz D, Elkihel M. Solving knapsack problems on GPU. *Computers & Operations Research* 2012; **39**(1):42–47.
12. Lalami ME, El-Baz D. GPU implementation of the branch and bound method for knapsack problems. *IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012*, IEEE, Shanghai, China, 2012; 1769–1777.
13. Pospíchal P, Schwarz J, Jaros J. Parallel genetic algorithm solving 0/1 knapsack problem running on the GPU. *16th International Conference on Soft Computing Mendel*, Brno University of Technology, Brno, Czech Republic, 2010; 64–70.
14. Ohshima S, Kise K, Katagiri T, Yuba T. Parallel processing of matrix multiplication in a CPU and GPU heterogeneous environment. In *High Performance Computing for Computational Science-VECPAR 2006*. Springer-Verlag: Berlin, 2007; 305–318.
15. Fatica M. Accelerating Linpack with CUDA on heterogeneous clusters. *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, ACM, 2009; 46–51.
16. Song FG, Tomov S, Dongarra J. Enabling and scaling matrix computations on heterogeneous multi-core and multi-GPU systems. *Proceedings of the 26th ACM International Conference on Supercomputing*, ACM, 2012; 365–376.
17. Ogata Y, Endo T, Maruyama N, Matsuoka S. An efficient, model-based CPU-GPU heterogeneous FFT library. In *IEEE International Symposium on Parallel and Distributed Processing, 2008. IPDPS 2008*, IEEE, 2008; 1–10.
18. Tomov S, Dongarra J, Baboulin M. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing* 2010; **36**(5):232–240.
19. Agullo E, Augonnet C, Dongarra J, Faverge M, Ltaief H, Thibault S, Tomov S. QR factorization on a multicore node enhanced with multiple GPU accelerators. *IEEE International Parallel & Distributed Processing Symposium (IPDPS), 2011*, IEEE, 2011; 932–943.
20. Chen RB, Tsai YM, Wang W. Adaptive block size for dense QR factorization in hybrid CPU-GPU systems via statistical modeling. *Parallel Computing* 2014; **40**(5):70–85.
21. Yu CD, Wang W, Pierce D. A CPU-GPU hybrid approach for the unsymmetric multifrontal method. *Parallel Computing* 2011; **37**(12):759–770.
22. Lu FS, Song JQ, Cao XQ, Zhu XQ. CPU/GPU computing for long-wave radiation physics on large GPU clusters. *Computers & Geosciences* 2012; **41**:47–55.
23. Wu Q, Yang CQ, Tang T, Xiao LQ. Exploiting hierarchy parallelism for molecular dynamics on a petascale heterogeneous system. *Journal of Parallel and Distributed Computing* 2013; **73**(12):1592–1604.
24. Lang J, Rüniger G. Dynamic distribution of workload between CPU and GPU for a parallel conjugate gradient method in an adaptive FEM. *Procedia Computer Science* 2013; **18**:299–308.
25. Lopez-Ortiz A, Salinger A, Suderman R. Toward a generic hybrid CPU-GPU parallelization of divide-and-conquer algorithms. *IEEE 27th International Parallel and Distributed Processing Symposium Workshops & PHD Forum (IPDPSW), 2013*, IEEE, 2013; 601–610.
26. Chakroun I, Melab N, Mezmaiz M, Tuytens D. Combining multi-core and GPU computing for solving combinatorial optimization problems. *Journal of Parallel and Distributed Computing* 2013; **73**(12):1563–1577.
27. Wan LJ, Li KL, Liu J, Li KQ. A novel CPU-GPU cooperative implementation of a parallel two-list algorithm for the subset-sum problem. *Proceedings of Programming Models and Applications on Multicores and Manycores*, ACM, 2014; 70–79.
28. Akl SG, Santoro N. Optimal parallel merging and sorting without memory conflicts. *IEEE Transactions on Computers* 1987; **100**(11):1367–1369.