CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2015; **27**:119–145
Published online 9 January 2014 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3201

GPU implementation of a parallel *two-list* algorithm for the subset-sum problem

Lanjun Wan [1], Kenli Li [1,2,*,†], Jing Liu [1] and Keqin Li [1,2,3]

[1] *College of Information Science and Engineering, Hunan University, Changsha, Hunan 410082, China*
[2] *National Supercomputing Center in Changsha, Changsha, Hunan 410082, China*
[3] *Department of Computer Science, State University of New York, New Paltz, New York 12561, USA*

SUMMARY

The subset-sum problem is a well-known non-deterministic polynomial-time complete (NP-complete) decision problem. This paper proposes a novel and efficient implementation of a parallel *two-list* algorithm for solving the problem on a graphics processing unit (GPU) using Compute Unified Device Architecture (CUDA). The algorithm is composed of a generation stage, a pruning stage, and a search stage. It is not easy to effectively implement the three stages of the algorithm on a GPU. Ways to achieve better performance, reasonable task distribution between CPU and GPU, effective GPU memory management, and CPU–GPU communication cost minimization are discussed. The generation stage of the algorithm adopts a typical recursive *divide-and-conquer* strategy. Because recursion cannot be well supported by current GPUs with compute capability less than 3.5, a new vector-based iterative implementation mechanism is designed to replace the explicit recursion. Furthermore, to optimize the performance of the GPU implementation, this paper improves the three stages of the algorithm. The experimental results show that the GPU implementation has much better performance than the CPU implementation and can achieve high speedup on different GPU cards. The experimental results also illustrate that the improved algorithm can bring significant performance benefits for the GPU implementation. Copyright © 2014 John Wiley & Sons, Ltd.

Received 12 May 2013; Revised 20 November 2013; Accepted 27 November 2013

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

1. INTRODUCTION

Given n positive integers W=[w_1, w_2, $\cdots$, w_n] and a positive integer M, the subset-sum problem (SSP) is the decision problem of finding a set $I \subseteq \{1, 2, \cdots, n\}$, 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, $\cdots$, x_n] for the equation

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

Subset-sum problem is well known to be NP-complete [1], and it is a special case of the 0/1 knapsack problem. It has many real-world applications, such as stock cutting, cargo loading, capital budgeting, job scheduling, workload allocation, and project selection [2–5]. The original search space of SSP has 2^n possible values. An exhaustive search would take $O(2^n)$ time to find a solution in the worst case. Many techniques have been developed to exactly solve SSP within a reasonable computation time. A branch-and-bound algorithm was proposed [6], and it achieves good performance for particular instances of SSP, but the worst-case time complexity is still $O(2^n)$. Another

*Correspondence to: Kenli Li, College of Information Science and Engineering, Hunan University, Changsha, Hunan 410082, China.
†E-mail: lkl510@263.net

Copyright © 2014 John Wiley & Sons, Ltd.

well-known approach is the dynamic programming algorithm [7], which solves SSP in pseudo-polynomial time, but it has exponential time complexity when the knapsack capacity is large. A tremendous improvement was made by Horowitz and Sahni [6], who developed a new technique that solves SSP in time $O(n2^{n/2})$ with $O(2^{n/2})$ memory space. The new technique is known as the *two-list* algorithm. On the basis of the *two-list* algorithm, Schroeppel and Shamir [8] proposed the *two-list four-table* algorithm, which needs the same time $O(n2^{n/2})$ and less memory space $O(2^{n/4})$ to solve SSP. Although many sequential algorithms have been designed to solve SSP in the past, Horowitz and Sahni's *two-list* algorithm continues to be the best known sequential algorithm.

With the advent of parallel computing, a large effort has been made to reduce the computation time of SSP. Karnin [9] proposed a parallel algorithm that parallelizes the generation routine of the *two-list four-table* algorithm [8] using $O(2^{n/6})$ processors and $O(2^{n/6})$ memory cells in time $O(n2^{n/2})$. Ferreira [10] presented a brilliant parallel *two-list* algorithm that solves SSP in time $O(n(2^{n/2})^{\varepsilon})$ with $O((2^{n/2})^{1-\varepsilon})$ processors and $O(2^{n/2})$ memory space, where $0 \leqslant \varepsilon \leqslant 1$. Chang *et al*. [11] introduced a parallel algorithm that parallelizes the generation stage of Horowitz and Sahni's *two-list* algorithm [6]. They claimed that their parallel generation stage can be accomplished in time $O((n/8)^2)$ with $O(2^{n/8})$ processors and $O(2^{n/4})$ memory space. On the basis of the *generation technique* of Chang *et al*., Lou and Chang [12] successfully parallelized the search stage of Horowitz and Sahni's $two\text{-}list$ algorithm [6] using $O(2^{n/8})$ processors and $O(2^{n/4})$ memory space in time $O(2^{3n/8})$. Unfortunately, the analysis of the time complexity of the algorithm of Chang *et al*., was proved to be wrong by Sanches *et al*. [13], and the correct time is bounded by $O(n2^{n/2})$. Sanches *et al*. [14] developed an optimal and scalable parallel *two-list* algorithm, which solves SSP in time $O(2^{n/2}/k)$ with $k = 2^q$ processors, where $0 \leqslant q \leqslant n/2 - 2\log n$. Chedid [15] proposed an optimal parallelization of Horowitz and Sahni's *two-list* algorithm [6], which solves SSP in time $O(2^{3n/8})$ with $O(2^{n/8})$ processors by combining Chedid's generation phase algorithm with Lou and Chang's search phase algorithm. Later, Chedid [16] presented another optimal and scalable parallel *two-list* algorithm, which solves SSP in time $O(2^{n/2-\alpha})$ with 2^{α} processors, where $0 \leqslant \alpha \leqslant n/2 - 2\log n + 2$.

The parallel algorithms described earlier are designed for the concurrent read exclusive write single instruction multiple data model with shared memory. On the basis of an exclusive read exclusive write single instruction multiple data model with shared memory, Li *et al*. [17] 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 $0 \leqslant \varepsilon \leqslant 1$.

Nowadays, graphics processing unit (GPU) computing is recognized as a powerful way to deal with time-intensive problems [18], because GPUs can offer high levels of data and thread parallelism, tremendous computational power, huge memory bandwidth, and comparatively low cost. In 2007, NVIDIA introduced the compute unified device architecture (CUDA) development environment as a general purpose parallel computing architecture, which makes GPU computing much more efficient and much easier for the programmer [19]. To implement parallel algorithms for solving the knapsack problems on a GPU, some work has been performed in recent years. Bokhari [20] explored a parallelization of the dynamic programming algorithm, which solves SSP on a 240-core NVIDIA FX 5800 GPU. Boyer *et al*. [21] proposed a parallel implementation of the dynamic programming method, which solves the 0-1 knapsack problem on a GPU. Lalami and El-Baz [22] presented an implementation of the branch-and-bound algorithm, which solves the 0-1 knapsack problem on a CPU–GPU system via CUDA. Pospíchal *et al*. [23] developed a parallel genetic algorithm, which accelerates the computation of the 0-1 knapsack problem on a GPU. However, to the authors' knowledge, there is no efficient parallel implementation of the *two-list* algorithm for solving SSP on a GPU.

In this paper, on the basis of the optimal parallel *two-list* algorithm of Li *et al*. [17], we propose a novel and efficient implementation of the algorithm for solving SSP on a GPU using CUDA. The GPU implementation of the algorithm is not straightforward and requires a significant effort. The main difficulties are as follows. (1) The algorithm is a typical example of an irregularly structured problem, so it is hard to implement the algorithm on a GPU. (2) The high CPU–GPU communication cost and memory access latency are the performance bottlenecks of the GPU implementation. (3) The generation stage of the algorithm adopts a typical recursive *divide-and-conquer*

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

strategy; however, recursion cannot be well supported by current GPUs with compute capability less than 3.5.

To effectively implement the algorithm on a GPU, reasonable task distribution between CPU and GPU, effective GPU memory management, and CPU–GPU communication cost minimization are discussed. A new vector-based iterative implementation mechanism is designed to replace the explicit recursion. In addition, to optimize the performance of the GPU implementation, the three stages of the algorithm are improved.

The rest of this paper is organized as follows. Section 2 gives a brief description of Horowitz and Sahni's *two-list* algorithm and the parallel *two-list* algorithm of Li *et al*., Section 3 presents an implementation of the parallel *two-list* algorithm on a GPU via CUDA. Section 4 describes the improved parallel *two-list* algorithm. Section 5 gives the experimental results and performance analysis. Section 6 concludes this paper and discusses the future work.

2. PREVIOUS WORK

In this section, we first briefly describe Horowitz and Sahni's sequential *two-list* algorithm [6] and the parallel *two-list* algorithm of Li *et al*. [17]. Then, we present the optimal parallel merging algorithm of Akl *et al*. [24]. In this way, our proposed GPU implementation becomes easier to understand.

2.1. The sequential two-list algorithm

This section presents Horowitz and Sahni's sequential *two-list* algorithm for solving SSP. The algorithm can be divided into two stages: (1) the generation stage, which is designed for generating two sorted lists and (2) the search stage, which is constructed to find a solution from the combinations of the two sorted lists. The sequential *two-list* algorithm is described as follows.

Generation stage:

1) Divide W into two equal parts: $W_1 = [w_1, w_2, \cdots, w_{n/2}]$ and W_2=$[w_{n/2+1}, w_{n/2+2}, \cdots, w_n]$.
2) Produce all $2^{n/2}$ possible subset sums of W_1, and then sort them in nondecreasing order and store them as the list A=$[a_1, a_2, \cdots, a_{2^{n/2}}]$.
3) Produce all $2^{n/2}$ possible subset sums of W_2, and then sort them in nonincreasing order and store them as the list $B = [b_1, b_2, \cdots, b_{2^{n/2}}]$.

Search stage:

1) (Initialize) $i = 1$, $j = 1$.
2) **If** $a_i + b_j = M$ **then stop**; a solution is found.
3) **If** $a_i + b_j < M$ **then** $i = i + 1$; **else** $j = j + 1$.
4) **If** $i > 2^{n/2}$ **or** $j > 2^{n/2}$ **then stop**; there is no solution.
5) Go to Step 2.

Before presenting the parallel *two-list* algorithm, some notation used in this algorithm is listed in Table I.

2.2. The parallel two-list algorithm

In this section, we introduce the parallel *two-list* algorithm proposed by Li *et al*. [17]. The algorithm contains three stages as follows: (1) the parallel generation stage, which is designed to generate two sorted lists; (2) the parallel pruning stage, which is designed to reduce the search overhead of each thread; and (3) the parallel search stage, which is designed to find a solution of SSP.

2.2.1. The generation stage. Because the procedures to generate two sorted lists A and B in reverse directions, nondecreasing and nonincreasing, are similar in essence, we only describe the procedure to generate the nondecreasing list A in the following presentation.

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

Table I. Notation used in the parallel *two-list* algorithm.

Notation	Explanation
W	A set of all the n items
W_1, W_2	Disjoint subsets of W of equal size $n/2$
n	The number of items in the set W
A	A sorted list in nondecreasing order
B	A sorted list in nonincreasing order
a_i	A subset of A
b_i	A subset of B
N	The number of subsets of A (or B), that is, $N = 2^{n/2}$
k	The number of required threads
e	The number of elements in each block, that is, $e = N/k$
M	The knapsack capacity

Algorithm 1 The parallel generation algorithm

Require: $A_1 = [0, w_1]$
1: **for** $i = 1$ **to** $n/2 - 1$ **do**
2: **if** $1 \leqslant i \leqslant n/4 - 1$ **then**
3: $k = 2^i$;
4: **else**
5: $k = 2^{n/4}$;
6: **end if**
7: **for all** k threads **do in parallel**
8: produce a new list A_i^1 by adding the item w_{i+1} to each element of the list A_i. Specifically, if $1 \leqslant i \leqslant n/4$, each thread works on one element of the list A_i; if $n/4 < i \leqslant n/2 - 1$, each thread works on $2^i/k$ elements of the list A_i;
9: use the optimal parallel merging algorithm to merge the two lists A_i and A_i^1 into a new nondecreasing list A_{i+1};
10: **end for**
11: **end for**
12: **return** $A_{n/2}$

To generate the nondecreasing list A, firstly, we initialize $A_1 = [0, w_1]$. Then, we use two threads to add the item w_2 to each element of the list A_1 in parallel, generating a new list $A_1^1 = [0 + w_2, w_1 + w_2]$. Specifically, each thread works on one element of the list A_1, that is, thread 1 adds w_2 to 0, and thread 2 adds w_2 to w_1. Furthermore, we use the optimal parallel merging algorithm [24], which will be described in Section 2.3 to merge the two lists A_1 and A_1^1 into a new nondecreasing list A_2. The aforementioned process is repeated until all the remaining items of W_1 have been considered. After processing the last item $w_{n/2}$, we obtain the final list $A_{n/2}$, that is, the nondecreasing list A. Suppose that the maximum number of threads used is $2^{n/4}$. When generating lists $A_2, A_3, \cdots, A_{n/4}$, only some of the threads are utilized; when generating lists $A_{n/4+1}, A_{n/4+2}, \cdots, A_{n/2}$, all threads must be utilized. The procedure to generate the nondecreasing list A is described in Algorithm 1. One can readily show that the running time of the parallel generation stage is $\sum_{i=1}^{n/4-1}(2 + (i+1)i) + \sum_{i=n/4}^{n/2-1}(2^{i+1}/(2k) + 2^{i+1}/k + \log k \times \log 2^{i+1}) = 3N/k$. Thus, the time complexity of the parallel generation algorithm is $O(3N/k)$.

2.2.2. The pruning stage. After the two sorted lists A and B have been generated, each is divided into k blocks, where each block contains $e = N/k$ elements. Suppose that the list A can be seen as $A = [\overline{A_1}, \overline{A_2}, \cdots, \overline{A_i}, \cdots, \overline{A_k}]$, where $\overline{A_i} = [\overline{a_{i.1}}, \overline{a_{i.2}}, \cdots, \overline{a_{i.e}}]$. Each element $\overline{a_{i.r}}$ in the sublist $\overline{A_i}$ represents a subset sum of A, where $1 \leqslant i \leqslant k$ and $1 \leqslant r \leqslant e$. The same convention is used for the list B. Then, the block $\overline{A_i}$ and the entire list B are assigned to thread P_i, where $1 \leqslant i \leqslant k$. Now,

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

if each thread starts to search the list B to find a solution, the running time is bounded by $O(N)$ in the worst case. To reduce the search overhead of each thread, it is necessary to find the *prune rule* to shrink the search space of each thread. Thus, the following two lemmas are introduced. Here, we do not provide the proof of these two lemmas, because the proof of these two lemmas is similar to that of Lemmas1 and 2 in [12].

Lemma 1
For any block pair $(\overline{A_i}, \overline{B_j})$, where $1 \leqslant i, j \leqslant k$, $\overline{A_i} = [\overline{a_{i.1}}, \overline{a_{i.2}}, \cdots, \overline{a_{i.e}}]$ and $\overline{B_j} = \left[\overline{b_{j.1}}, \overline{b_{j.2}}, \cdots, \overline{b_{j.e}}\right]$, if $\overline{a_{i.1}} + \overline{b_{j.e}} > M$, then any element pair $\left(\overline{a_{i.r}}, \overline{b_{j.s}}\right)$ is not a solution of SSP, where $\overline{a_{i.r}} \in \overline{A_i}$ and $\overline{b_{j.s}} \in \overline{B_j}$.

Lemma 2
For any block pair $(\overline{A_i}, \overline{B_j})$, where $1 \leqslant i, j \leqslant k$, $\overline{A_i} = [\overline{a_{i.1}}, \overline{a_{i.2}}, \cdots, \overline{a_{i.e}}]$ and $\overline{B_j} = \left[\overline{b_{j.1}}, \overline{b_{j.2}}, \cdots, \overline{b_{j.e}}\right]$, if $\overline{a_{i.e}} + \overline{b_{j.1}} < M$, then any element pair $\left(\overline{a_{i.r}}, \overline{b_{j.s}}\right)$ is not a solution of SSP, where $\overline{a_{i.r}} \in \overline{A_i}$ and $\overline{b_{j.s}} \in \overline{B_j}$.

On the basis of Lemmas 1 and 2, we design the parallel pruning algorithm described in Algorithm 2, which is performed before the search routine in order to shrink the search space of each thread. For the case in Lemma 1 or 2, when any element pair $\left(\overline{a_{i.r}}, \overline{b_{j.s}}\right)$ of block pair $(\overline{A_i}, \overline{B_j})$ is not a solution of SSP, the block pair $(\overline{A_i}, \overline{B_j})$ is pruned. Once one block pair has been picked, it is written into the shared memory to check it in the next search stage. Because each thread prunes its individual search space, one can readily see that the time complexity of the parallel pruning algorithm is $O(k)$. 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 proof of this fact is given in [17].

2.2.3. The search stage. After the pruning stage has completed, at most $2k - 1$ picked block pairs are picked. In the search stage, these picked block pairs are evenly assigned to k threads. Obviously, each thread has at most two block pairs to be searched. Each thread performs the search routine of Horowitz and Sahni's *two-list* algorithm in parallel, so as to find a solution of SSP. The search routine takes $O(6N/k)$ time to find a solution in the worst case.

Combining the parallel generation algorithm and the parallel pruning algorithm together, the complete parallel *two-list* algorithm for the SSP is shown as follows.

Step 1: Perform the parallel generation algorithm.
Step 2: Perform the parallel pruning algorithm.

Algorithm 2 The parallel pruning algorithm

Require: The two sorted lists A and B are evenly divided into k blocks, respectively.
1: **for all** $P_i, 1 \leqslant i \leqslant k$ **do in parallel**
2: **for** $j = i$ **to** $k + i - 1$ **do**
3: $X = \overline{a_{i.1}} + \overline{b_{(j \bmod k).e}}$;
4: $Y = \overline{a_{i.e}} + \overline{b_{(j \bmod k).1}}$;
5: **if** $X = M$ **or** $Y = M$ **then**
6: **stop**; ▷ a solution is found
7: **else if** $X < M$ **and** $Y > M$ **then**
8: write $(\overline{A_i}, \overline{B_{j \bmod k}})$ to the shared memory;
9: **end if**
10: **end for**
11: **end for**
12: **return** a solution **or** all the picked block pairs

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

Step 3: Evenly assign the picked block pairs to k threads.
Step 4: For all P_i where $1 \leqslant i \leqslant k$ do in parallel

Thread P_i performs the search routine of Horowitz and Sahni's *two-list* algorithm.

2.3. The optimal parallel merging algorithm

This section introduces the optimal parallel merging algorithm without memory conflicts for the exclusive read exclusive write parallel random access machine proposed by Akl *et al*. [24]. Here, for the sake of consistency, we describe the merging algorithm with a slight modification. Given two sorted vectors $U = [u_1, u_2, \cdots, u_m]$ and $V = [v_1, v_2, \cdots, v_m]$, according to the following steps, U and V can be merged without memory conflicts into a new vector of length $2m$.

Step 1: Use k threads to divide U and V each into k subvectors $[U_1, U_2, \cdots, U_k]$ and $[V_1, V_2, \cdots, V_k]$ in parallel, $|U_i| + |V_i| = 2m/k$, for $1 \leqslant i \leqslant k$, and all elements in U_i and V_i are smaller than those of all elements in U_{i+1} and V_{i+1}, for $1 \leqslant i \leqslant k-1$, where $1 \leqslant k \leqslant 2m$ and k is a power of 2.
Step 2: Use thread P_i to merge U_i and V_i, for $1 \leqslant i \leqslant k$.

Step 1 can be efficiently implemented using the selection algorithm presented in [24]. The total time required for Step 1 is $O(\log k \times \log 2m)$; Step 2 requires each thread to merge at most $2m/k$ elements; therefore, the total time complexity of this parallel merging algorithm is $O(2m/k + \log k \times \log 2m)$.

3. THE PROPOSED GPU IMPLEMENTATION

In this section, firstly, we briefly introduce the CUDA architecture. Then, on the basis of the parallel *two-list* algorithm of Li *et al*. [17], we describe how to effectively implement the three stages of the algorithm on a GPU.

3.1. The CUDA architecture

This section gives a brief description of the CUDA architecture, which is based on the single instruction multiple threads mode of programming, as shown in Figure 1.

Parallel code executed on the GPU (the so-called device) is interleaved with serial code executed on the CPU (the so-called host). A thread block is a batch of threads that can cooperate by sharing

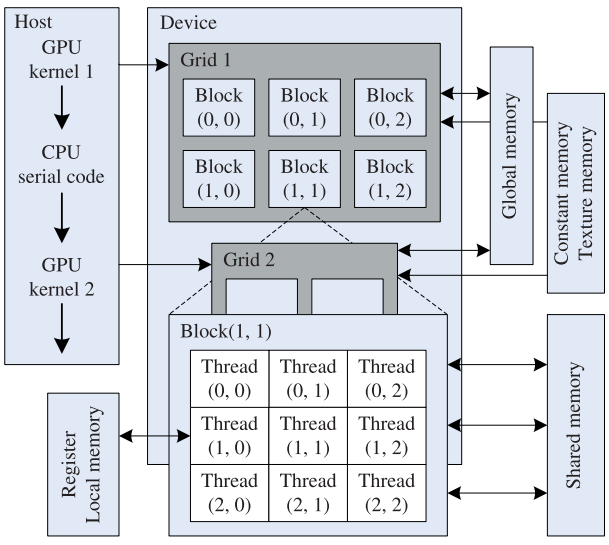


Figure 1. The CUDA architecture.

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

data through shared memory. Threads from different blocks cannot cooperate. These thread blocks are organized into a grid. The basic unit of execution in CUDA is the so-called kernel. When a CUDA program invokes a kernel on the host side, these thread blocks within a grid are enumerated and distributed to multiprocessors with available execution capacity. All threads within a grid can be executed in parallel.

As can be seen in Figure 1, the GPU offers multiple memory spaces. Each thread has its own register and local memory. Each thread block has a shared memory, which is visible to all the threads within the same block and has the same lifetime as the block. All threads within a grid can access the global memory, the constant memory, and the texture memory. These threads can coordinate memory accesses by synchronizing their executions.

3.2. Task distribution

This section describes how to reasonably assign tasks to CPU and GPU in our GPU implementation.

A reasonable task distribution between CPU and GPU is critical for GPU applications, namely, one has to identify what is suitable to be performed on the CPU or on the GPU. In our GPU implementation, we assign most tasks to the GPU, that is, the three stages of the algorithm are executed on the GPU, whereas CPU is only responsible for controlling the whole running process of this algorithm and transferring a small amount of data. Task distribution between CPU and GPU is shown in Figure 2.

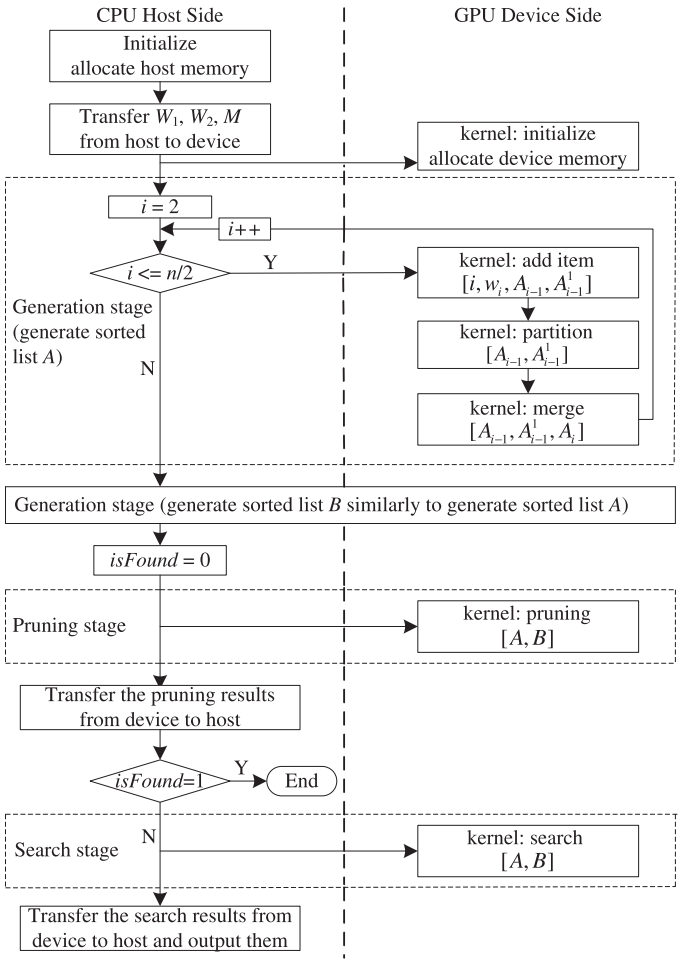


Figure 2. Task distribution between CPU and GPU.

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

Specifically, our GPU implementation begins with the initialization on the host and device sides. During the host side initialization phase, firstly, we obtain the original n-element input vector W, divide W into two equal parts W_1 and W_2, and dynamically allocate host memory and global memory for W_1 and W_2, where each part contains $n/2$ items. Secondly, we statically allocate constant memory for the knapsack capacity M on the device side, and we initialize the capacity value. Furthermore, we transfer W_1, W_2, and M from the host to the device through the system bus. W_1 and W_2 are stored in the global memory. M is copied to the constant memory, and it can be read by all the threads in the grid with lower memory access latency than the global memory.

In our GPU implementation, we use the list A to store all of the N subset sums of W_1 in nondecreasing order, and we use the list B to store all of the N subset sums of W_2 in nonincreasing order. During the device side initialization phase, we dynamically allocate global memory for lists A and B by executing the initialize kernel, where each list needs $O(N)$ memory cells. In addition, we initialize $A_1 = [0, w_1]$ and $B_1 = [w_{n/2+1}, 0]$.

After the initialization phase has completed, we will describe how to execute the three stages of the algorithm on a GPU in the following sections.

3.3. The parallel generation stage

This section describes the generation procedure of the nondecreasing list A. The generation procedure of the nonincreasing list B is almost the same as that of the list A.

Let us suppose that the list $A_1 = [0, w_1]$ has been generated in the previous initialization phase. Firstly, we use k GPU threads to add the item w_i to each element of the list $A_{i-1} = [a_{i-1.1}, a_{i-1.2}, \cdots, a_{i-1.2^{i-1}}]$ in parallel, generating a new list $A^1_{i-1} = [a_{i-1.1} + w_i, a_{i-1.2} + w_i, \cdots, a_{i-1.2^{i-1}} + w_i]$, where $2 \leqslant i \leqslant n/2$. Suppose that the maximum number of GPU threads used is $2^{n/4}$. If $2 \leqslant i \leqslant n/4 + 1$, each thread works on one element of the list A_{i-1}; if $n/4 + 1 < i \leqslant n/2$, each thread works on $2^{i-1}/k$ elements of the list A_{i-1}. Note that it is necessary to load the item w_i from global memory to on-chip shared memory, before the add operation is executed by each thread, so as to avoid the high global memory access latency. Then, we merge lists A_{i-1} and A^1_{i-1} into a new list A_i with 2^i subset sums by using the optimal parallel merging algorithm [24], and write A_i into the global memory. By sequentially repeating such steps, after $n/2 - 1$ iterations, we finally obtain the list $A_{n/2}$ with N subset sums, that is, the needed sorted list A.

The basic procedure to generate the nondecreasing list A is described in Algorithm 3, which shows that the task carried out by the CPU on the host side has three phases: the add item phase, the partition phase, and the merge phase. The three phases correspond to three different kernels carried out by the GPU on the device side as follows: the add item kernel, the partition kernel, and the merge kernel. When these kernels are launched on the host side, we must specify the number of blocks per grid and the number of threads per block (TPB) in *gridSize* and *blockSize*, respectively. Between the three phases, we use the function *cudaDeviceSynchronize*() to synchronize the GPU threads between two kernel launches, so as to insure data consistency.

From Algorithm 3, it is easy to see that when generating the list A_i, if $2 \leqslant i \leqslant n/4$, the number of required threads is 2^{i-1} at most; if $n/4 < i \leqslant n/2$, the number of threads used is $2^{n/4}$. Hence, we can determine the number of required threads according to the size of the list A_i. Moreover, the communication cost between CPU and GPU can be minimized, because the list A is directly generated by executing the three kernels on the GPU, and it is directly stored in the global memory.

3.3.1. The add item phase. In the add item phase, the add item kernel is described in Algorithm 4, which displays the work of a single GPU thread. Here, all k GPU threads can run the kernel in parallel. The *tid*-th thread adds the item w_i to the *tid*-th element of the list A_{i-1}, generating the corresponding *tid*-th element of the list A^1_{i-1}, where $1 \leqslant tid \leqslant 2^{i-1}$.

In Algorithm 4, *gridDim.x* indicates the number of blocks in a grid in the X-direction; *blockDim.x* indicates the number of threads in a block in the X-direction; *blockIdx.x* represents the block index within a grid; and *threadIdx.x* represents the thread index within a block. In our GPU implementation, we use a one-dimensional block and grid. Thus *blockDim.x* $*$ *gridDim.x* is the total number of

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

Algorithm 3 The basic procedure to generate the nondecreasing list A

Require: $W_1 = [w_1, w_2, \cdots, w_{n/2}], A_1 = [0, w_1]$
1: **for** $i = 2$ **to** $n/2$ **do**
2: **if** $2 \leqslant i \leqslant n/4$ **then**
3: $k = 2^{i-1}$;
4: **else**
5: $k = 2^{n/4}$;
6: **end if**
7: *addItem_kernel*$<<< gridSize, blockSize >>> \left(i, w_i, A_{i-1}, A_{i-1}^1\right)$;
8: *cudaDeviceSynchronize*();
9: **for** $j = 1$ **to** $\log k$ **do**
10: *partition_kernel*$<<< gridSize, blockSize >>> \left(j, A_{i-1}, A_{i-1}^1\right)$;
11: *cudaDeviceSynchronize*();
12: **end for**
13: *merge_kernel*$<<< gridSize, blockSize >>> \left(A_{i-1}, A_{i-1}^1, A_i\right)$;
14: *cudaDeviceSynchronize*();
15: **end for**
16: **return** $A_{n/2}$

Algorithm 4 The add item kernel

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

threads running in the grid. Each thread evaluates one copy of the kernel. After each thread accomplishes its work at the current index *tid*, if *tid* is less than or equal to 2^{i-1}, we need to increase *tid* by *blockDim.x* $*$ *gridDim.x*.

3.3.2. The partition phase. The optimal parallel merging algorithm [24] is used in the partition and merge phases. It is clear that the partition process adopts a typical recursive *divide-and-conquer* strategy. However, support for recursion in GPUs with compute capability less than 3.5 is weak. Specifically, they support recursion only for device functions but not for kernel functions, namely they do not allow a kernel function to call other kernel functions recursively, and thus massive GPU cores cannot be utilized, leading to poor efficiency. As a consequence, we have to develop a new vector-based iterative implementation mechanism instead of the explicit recursion. The partition and merge phases are illustrated in Figure 3.

As shown in Figure 3, the whole partition process needs to execute $\log k$ iterations to complete. In each iteration, the partition kernel is launched from the host side and executed on the device side; the partition kernel is described in Algorithm 5. In the partition phase, the number of required GPU threads depends on the current iteration number j, where $1 \leqslant j \leqslant \log k$. Let *threadCounts* denote the number of required GPU threads, which is equal to 2^{j-1} for the j-th iteration. Note that the current iteration number j should be copied from the host to the device and stored in the on-chip shared memory, before the partition kernel is launched.

In the partition phase, in order to store the partition information (i.e., boundary positions of sublist) generated by each GPU thread in each iteration, we declare a vector H whose size is $2k - 1$. The vector H is directly defined in the global memory, which can contribute to greatly reducing the

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

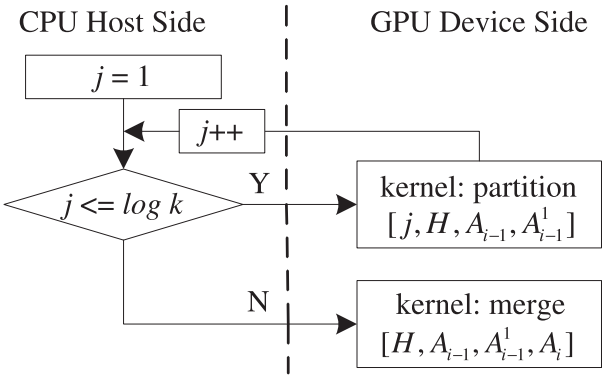


Figure 3. The partition and merge phases.

Algorithm 5 The partition kernel

Require: j, A_{i-1}, A^1_{i-1}, H
1: $tid = blockIdx.x * blockDim.x + threadIdx.x + 1$;
2: $threadCounts = 2^{j-1}$;
3: **while** $tid \leqslant threadCounts$ **do**
4: $medianId = tid + threadCounts - 1$;
5: $a = H[medianId].low_A$;
6: $b = H[medianId].high_A$;
7: $c = H[medianId].low_B$;
8: $d = H[medianId].high_B$;
9: find the median pair (e, f) of $A_{i-1}[a, b]$ and $A^1_{i-1}[c, d]$ by using the $selection\ algorithm$;
10: $H[2 * medianId].low_A = a$;
11: $H[2 * medianId].high_A = e$;
12: $H[2 * medianId].low_B = c$;
13: $H[2 * medianId].high_B = f$;
14: $H[2 * medianId + 1].low_A = e + 1$;
15: $H[2 * medianId + 1].high_A = b$;
16: $H[2 * medianId + 1].low_B = f + 1$;
17: $H[2 * medianId + 1].high_B = d$;
18: $tid += blockDim.x * gridDim.x$;
19: **end while**
20: **return** H

communication cost between CPU and GPU, because the partition information does not have to be passed back and forth to the GPU in each iteration.

Each element of the vector H is a struct with four integer member variables: low_A, $high_A$, low_B and $high_B$. The member variables low_A and $high_A$ are used to store the lower bound position and the upper bound position of sublist X, respectively. Similarly, the member variables low_B and $high_B$ are used to store the lower bound position and the upper bound position of sublist Y, respectively. Before the first iteration, in order to store the boundary positions of lists A_{i-1} and A^1_{i-1}, we initialize H as follows: $H[1].low_A = 1$, $H[1].high_A = 2^{i-1}$, $H[1].low_B = 1$ and $H[1].high_B = 2^{i-1}$.

Here, let $A_{i-1}[a, b]$ denote the sublist $[a_{i-1.a}, a_{i-1.a+1}, \cdots, a_{i-1.b}]$, and $A^1_{i-1}[c, d]$ denote the sublist $\left[a^1_{i-1.c}, a^1_{i-1.c+1}, \cdots, a^1_{i-1.d}\right]$, where $2 \leqslant i \leqslant n/2$ and $1 \leqslant a, b, c, d \leqslant 2^{i-1}$. The implementation of partitioning lists A_{i-1} and A^1_{i-1} consists of the following $\log k$ steps:

Step 1: $(j = 1)$ We only use a single GPU thread P_1 to run the partition kernel. At first, P_1 gets the first element of H. Then, P_1 executes the selection algorithm presented in [24]

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

to find the median pair (x, y) of lists $A_{i-1}[1, 2^{i-1}]$ and $A^1_{i-1}[1, 2^{i-1}]$. Finally, we use the second element of H to store the boundary positions $(1, x, 1, y)$ and use the third element of H to store the boundary positions $(x + 1, 2^{i-1}, y + 1, 2^{i-1})$.

Step j: $(2 \leqslant j \leqslant \log k)$ We use 2^{j-1} GPU threads to run the partition kernel in parallel. Firstly, the current thread P_{tid} gets the corresponding element of H via the variable *medianId*, namely, it gets the corresponding partition information (a, b, c, d) from the previous iteration, where $1 \leqslant tid \leqslant 2^{j-1}$. Secondly, P_{tid} finds the median pair (e, f) of sublists $A_{i-1}[a, b]$ and $A^1_{i-1}[c, d]$. Finally, the new boundary positions (a, e, c, f) are stored in the $(2 * medianId)$-th element of H. In the same way, the new boundary positions $(e + 1, b, f + 1, d)$ are stored in the $(2 * medianId + 1)$-th element of H.

3.3.3. The merge phase. In the merge phase, we use k GPU threads to carry out the merge kernel described in Algorithm 6 in parallel. To begin with, the current thread P_{tid} obtains the corresponding partition information (a, b, c, d) from the last iteration of the aforementioned partition phase through *medianId* and H, where $1 \leqslant tid \leqslant k$. After that, P_{tid} merges lists $A_{i-1}[a, b]$ and $A^1_{i-1}[c, d]$ into the list A_i, and the result of merging is placed in locations $a + c - 1$ to $b + d$ in the list A_i.

3.4. The parallel pruning stage

This section describes how to shrink the search space of SSP efficiently, so as to reduce the running time of the search stage.

Similar to the pruning stage described in Section 2.2.2, we first evenly divide lists A and B into k blocks of e elements. For simplicity, let $A = \left[\overline{A_1}, \overline{A_2}, \cdots, \overline{A_i}, \cdots, \overline{A_k}\right]$ and $B = \left[\overline{B_1}, \overline{B_2}, \cdots, \overline{B_j}, \cdots, \overline{B_k}\right]$, where $\overline{A_i} = [\overline{a_{i.1}}, \overline{a_{i.2}}, \cdots, \overline{a_{i.e}}]$, $\overline{B_j} = \left[\overline{b_{j.1}}, \overline{b_{j.2}}, \cdots, \overline{b_{j.e}}\right]$, $1 \leqslant i, j \leqslant k$. Then, we assign the block $\overline{A_i}$ and the entire list B to the GPU thread P_i, where $1 \leqslant i \leqslant k$. Finally, we use k GPU threads to carry out the pruning kernel described in Algorithm 7 in parallel.

During the pruning process, once any block pair $\left(\overline{A_i}, \overline{B_j}\right)$ is picked by the thread P_i, we should use P_i to write the block indices i and j into a vector, where $1 \leqslant i, j \leqslant k$. We can evenly allocate those picked block pairs to k GPU threads through the vector, in the next search stage.

Before the pruning kernel is executed, we declare a vector S whose size is $2k - 1$, and dynamically allocate global memory for it. Each element of the vector S is a struct with two integer member variables *bidA* and *bidB*, which are used to record the picked block index within lists A and B, respectively. In addition, we declare two device variables r and *isFound* in the global memory; r is used as a counter to count the number of block pairs to be picked, and *isFound* is used to determine whether a solution has been found. We initialize r and *isFound* to zero, respectively.

During the pruning kernel execution, when the thread P_i searches the block pair $\left(\overline{A_i}, \overline{B_j}\right)$, where $1 \leqslant i, j \leqslant k$, if $\overline{a_{i.1}} + \overline{b_{j.e}} = M$ or $\overline{a_{i.e}} + \overline{b_{j.1}} = M$, then a solution has been found and the variable

Algorithm 6 The merge kernel

Require: A_{i-1}, A^1_{i-1}, H, k
1: $tid = blockIdx.x * blockDim.x + threadIdx.x + 1;$
2: **while** $tid \leqslant k$ **do**
3: $\quad medianId = tid + k - 1;$
4: $\quad a = H[medianId].low_A;$
5: $\quad b = H[medianId].high_A;$
6: $\quad c = H[medianId].low_B;$
7: $\quad d = H[medianId].high_B;$
8: $\quad$ merge $A_{i-1}[a, b]$ and $A^1_{i-1}[c, d]$ into $A_i[a + c - 1, b + d];$
9: $\quad tid += blockDim.x * gridDim.x;$
10: **end while**
11: **return** A_i

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

isFound is set to 1; if $\overline{a_{i.1}} + \overline{b_{j.e}} < M$ and $\overline{a_{i.e}} + \overline{b_{j.1}} > M$, then the block pair $\left(\overline{A_i}, \overline{B_j}\right)$ is picked, and the block indices i and j will be stored in the r-th element of S; otherwise, the block pair $\left(\overline{A_i}, \overline{B_j}\right)$ is directly discarded. Note that *isFound* and r are global variables, namely they might be read and changed simultaneously by all k threads. Hence, it is necessary to update them atomically.

After the pruning kernel has completed, if *isFound*=1, we copy the solution from the device to the host and output it; otherwise, we will carry out the next search stage.

3.5. The parallel search stage

This section describes how to search a solution of SSP efficiently by performing the search routine of Horowitz and Sahni's *two-list* algorithm in parallel.

Similar to the search stage described in Section 2.2.3, we first evenly assign those picked block pairs to k GPU threads. Because the pruning stage picks at most $2k-1$ block pairs, each thread will search two block pairs at most. For clarity, let us suppose that the block pair $\left(\overline{A_s}, \overline{B_t}\right)$ is assigned to the thread P_i, where $\overline{A_s} = [\overline{a_{s.1}}, \overline{a_{s.2}}, \cdots, \overline{a_{s.e}}]$, $\overline{B_t} = \left[\overline{b_{t.1}}, \overline{b_{t.2}}, \cdots, \overline{b_{t.e}}\right]$ and $1 \leqslant i, s, t \leqslant k$. Then, we use k GPU threads to perform the search kernel described in Algorithm 8 in parallel.

During the search kernel execution, the thread P_i firstly obtains the indices (s, t) of the block pair assigned to itself. Then, P_i finds the top elements of the block pair $\left(\overline{A_s}, \overline{B_t}\right)$, and if $\overline{a_{s.1}} + \overline{b_{t.1}} = M$, then a solution has been found and the variable *isFound* is set to 1; if $\overline{a_{s.1}} + \overline{b_{t.1}} < M$, 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.

After the search kernel has completed, if *isFound*=1, we copy the solution from the device to the host and output it; otherwise, there is no solution.

4. PERFORMANCE OPTIMIZATION

To optimize the performance of the proposed GPU implementation, on the basis of the parallel *two-list* algorithm of Li *et al*. [17], we describe how to improve the three stages of the algorithm in this section.

Algorithm 7 The pruning kernel

Require: $A, B, M, S, r = 0, isFound = 0, e = N/k$

```
1:  i = blockIdx.x * blockDim.x + threadIdx.x + 1;
2:  while i ⩽ k do
3:    for j = i to k + i − 1 do
4:      if isFound = 1 then
5:        break; ▷ a solution is found
6:      else
7:        X = a_{i.1} + b_{(j mod k).e};
8:        Y = a_{i.e} + b_{(j mod k).1};
9:        if X = M or Y = M then
10:         atomicExch(&isFound, 1);
11:       else if X < M and Y > M then
12:         atomicAdd(&r, 1);
13:         S[r].bidA = i;
14:         S[r].bidB = j mod k;
15:       end if
16:     end if
17:   end for
18:   i += blockDim.x * gridDim.x;
19: end while
20: return a solution or S
```

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

Algorithm 8 The search kernel

Require: $A, B, M, S, r, isFound, e = N/k$

```
1: i = blockIdx.x * blockDim.x + threadIdx.x + 1;
2: while i ⩽ r do
3:    s = S[i].bidA;
4:    t = S[i].bidB;
5:    x = 1;
6:    y = 1;
7:    while x ⩽ e and y ⩽ e do
8:       if isFound = 1 then
9:          break; ▷ a solution is found
10:      else
11:         if a_{s.x} + b_{t.y} = M then
12:            atomicExch(&isFound, 1);
13:         else if a_{s.x} + b_{t.y} < M then
14:            x = x + 1;
15:         else
16:            y = y + 1;
17:         end if
18:      end if
19:   end while
20:   i += blockDim.x * gridDim.x;
21: end while
22: return a solution or NULL
```

4.1. The improved generation stage

To improve the performance of the generation stage, during the process of generating the nondecreasing list A, we discard those subset sums that exceed the knapsack capacity.

Specifically, we first sort the input vector W in nonincreasing order and then divide it into two equal parts W_1 and W_2. When adding the item w_i to each element of the list $A_{i-1} = [a_{i-1.1}, a_{i-1.2}, \cdots, a_{i-1.2^{i-1}}]$ to generate a new list $A^1_{i-1} = [a_{i-1.1} + w_i, a_{i-1.2} + w_i, \cdots, a_{i-1.2^{i-1}} + w_i]$, if $a_{i-1.j} + w_i > M$, the subset sum can be directly discarded, namely it does not need to be stored in the list A^1_{i-1}, where $2 \leqslant i \leqslant n/2$ and $1 \leqslant j \leqslant 2^{i-1}$.

According to this optimization strategy, the improved procedure to generate the nondecreasing list A is described in Algorithm 9. Let D and D^1 denote, respectively, the number of elements to be processed in the lists A_{i-1} and A^1_{i-1}, where $2 \leqslant i \leqslant n/2$. Both D and D^1 are initialized to 2. Let DS_{gpu} and DS_{cpu} denote the number of subset sums to be discarded in the list A^1_{i-1} in each iteration. Note that DS_{gpu} is a device variable declared in the global memory. The whole process of generating the list A needs to execute $n/2 - 1$ iterations to complete. Each iteration consists of the following five steps:

Step 1: Assign 0 to DS_{gpu} on the device side.

Step 2: Use k GPU threads to execute the improved add item kernel described in Algorithm 10 in parallel. If $A_{i-1}[tid] + w_i > M$, then the subset sum does not need to be stored in the list A^1_{i-1}, and the value of variable DS_{gpu} needs to be increased by 1, where $1 \leqslant tid \leqslant D$.

Step 3: Transfer the value of variable DS_{gpu} from the device to the host, and store it into variable DS_{cpu}, after completing the add item process.

Step 4: Use k GPU threads to merge A_{i-1} and A^1_{i-1} into a new list A_i in parallel. Because DS_{cpu} subset sums are discarded in the list A^1_{i-1}, the number of elements to be merged in the list A^1_{i-1} is $D^1 = D - DS_{cpu}$. Accordingly, during the partition phase, the initial boundary positions of lists A_{i-1} and A^1_{i-1} should be updated as follows: $H[1].low_A = 1, H[1].high_A = D, H[1].low_B = 1, H[1].high_B = D^1$.

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

Algorithm 9 The improved procedure to generate the nondecreasing list A

Require: $W_1 = [w_1, w_2, \cdots, w_{n/2}]$, $A_1 = [0, w_1]$, $D = 2$, $D^1 = 2$, $DS_{gpu} = 0$, $DS_{cpu} = 0$
1: **for** $i = 2$ **to** $n/2$ **do**
2: **if** $2 \leqslant i \leqslant n/4$ **then**
3: $k = 2^{i-1}$;
4: **else**
5: $k = 2^{n/4}$;
6: **end if**
7: assign 0 to DS_{gpu} on the device side;
8: *addItem_kernel*$<<< gridSize, blockSize >>> (D, w_i, A_{i-1}, A^1_{i-1})$;
9: *cudaDeviceSynchronize*();
10: *copyFromGPUToCPU*(DS_{cpu}, DS_{gpu});
11: $D^1 = D - DS_{cpu}$;
12: **for** $j = 1$ **to** $\log k$ **do**
13: *partition_kernel*$<<< gridSize, blockSize >>> (j, A_{i-1}, A^1_{i-1})$;
14: *cudaDeviceSynchronize*();
15: **end for**
16: *merge_kernel*$<<< gridSize, blockSize >>> (A_{i-1}, A^1_{i-1}, A_i)$;
17: *cudaDeviceSynchronize*();
18: $D = D + D^1$;
19: **end for**
20: **return** $A_{n/2}$

Algorithm 10 The improved add item kernel

Require: $D, w_i, A_{i-1}, A^1_{i-1}$
1: $tid = blockIdx.x * blockDim.x + threadIdx.x + 1$;
2: **while** $tid \leqslant D$ **do**
3: **if** $A_{i-1}[tid] + w_i > M$ **then**
4: $atomicAdd(\&DS_{gpu}, 1)$;
5: **else**
6: $A^1_{i-1}[tid] = A_{i-1}[tid] + w_i$;
7: **end if**
8: $tid += blockDim.x * gridDim.x$;
9: **end while**
10: **return** A^1_{i-1}

Step 5: Update the value of variable D as follows: $D = D + D^1$, before the next iteration, because the number of elements in the new list A_i is $D + D^1$.

In our improved generation stage (IGS), those subset sums that exceed the knapsack capacity are discarded in each iteration rather than after the entire list A has been generated. It is evident that the smaller the knapsack capacity is, the more subset sums can be discarded. Therefore, the IGS can give better performance than the original generation stage (OGS) for small knapsack capacity.

4.2. The improved pruning stage

To improve the performance of the pruning stage, on the basis of Lemmas 1 and 2, we introduce the following Lemmas 3 and 4 to reduce the pruning time of each thread.

Lemma 3
For any block $\overline{A_i}$ and list B, $\overline{A_i} = (\overline{a_{i.1}}, \overline{a_{i.2}}, \cdots, \overline{a_{i.e}})$, $B = \left[\overline{B_1}, \overline{B_2}, \cdots, \overline{B_{low}}, \cdots, \overline{B_k}\right]$, where

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

$\overline{B_{low}} = \left[\overline{b_{low.1}}, \overline{b_{low.2}}, \cdots, \overline{b_{low.e}}\right]$, $1 \leqslant i, low \leqslant k$, if $\overline{a_{i.1}} + \overline{b_{low-1.e}} > M$ and $\overline{a_{i.1}} + \overline{b_{low.e}} < M$, then any block pair $\left(\overline{A_i}, \overline{B_j}\right)$ can be discarded, where $1 < low \leqslant k$ and $1 \leqslant j < low$.

Proof
It is obvious that the element $\overline{a_{i.1}}$ is the smallest element in the block $\overline{A_i}$ and the element $\overline{b_{low-1.e}}$ is the smallest element in the block $\overline{B_{low-1}}$. If the sum of these two elements is greater than M, then $\overline{a_{i.1}} + \overline{b_{j.e}} > M$, where $1 \leqslant j \leqslant low - 1$. That is, any block pair $\left(\overline{A_i}, \overline{B_j}\right)$ can be discarded. □

Lemma 4
For any block $\overline{A_i}$ and list B, $\overline{A_i} = (\overline{a_{i.1}}, \overline{a_{i.2}}, \cdots, \overline{a_{i.e}})$, $B = \left[\overline{B_1}, \overline{B_2}, \cdots, \overline{B_{high}}, \cdots, \overline{B_k}\right]$, where $\overline{B_{high}} = \left[\overline{b_{high.1}}, \overline{b_{high.2}}, \cdots, \overline{b_{high.e}}\right]$, $1 \leqslant i, high \leqslant k$, if $\overline{a_{i.e}} + \overline{b_{high.1}} > M$ and $\overline{a_{i.e}} + \overline{b_{high+1.1}} < M$, then any block pair $\left(\overline{A_i}, \overline{B_j}\right)$ can be discarded, where $1 \leqslant high < k$ and $high < j \leqslant k$.

Proof
It is obvious that the element $\overline{a_{i.e}}$ is the largest element in the block $\overline{A_i}$ and the element $\overline{b_{high+1.1}}$ is the largest element in the block $\overline{B_{high+1}}$. If the sum of these two elements is smaller than M, then $\overline{a_{i.e}} + \overline{b_{j.1}} < M$, where $high + 1 \leqslant j \leqslant k$. That is, any block pair $\left(\overline{A_i}, \overline{B_j}\right)$ can be discarded. □

On the basis of Lemmas 3 and 4, we use GPU thread P_i to perform two binary searches in the list B to find the block indices *low* and *high*, so as to pick $high - low + 1$ consecutive blocks $\overline{B_{low}}$, $\overline{B_{low+1}}$, $\cdots$, $\overline{B_{high}}$, which will be associated with block A_i, where $1 \leqslant i \leqslant k$. The improved pruning kernel is described in Algorithm 11, which consists of the following three steps:

Algorithm 11 The improved pruning kernel

Require: $A, B, M, S, r = 0, isFound = 0, e = N/k$
1: $i = blockIdx.x * blockDim.x + threadIdx.x + 1$;
2: **while** $i \leqslant k$ **do**
3: $l = 1; r = k$;
4: **while** $l \leqslant r$ **and** $isFound = 0$ **do**
5: $m = l + (r - l)/2$;
6: **if** $\overline{a_{i.1}} + \overline{b_{m.e}} = M$ **then** $atomicExch(\&isFound, 1)$; break; ▷ a solution is found
7: **else if** $\overline{a_{i.1}} + \overline{b_{m.e}} < M$ **then** $r = m - 1$; **else** $l = m + 1$; **end if**
8: **end while**
9: $low = l; r = k$;
10: **while** $l \leqslant r$ **and** $isFound = 0$ **do**
11: $m = l + (r - l)/2$;
12: **if** $\overline{a_{i.e}} + \overline{b_{m.1}} = M$ **then** $atomicExch(\&isFound, 1)$; break; ▷ a solution is found
13: **else if** $\overline{a_{i.e}} + \overline{b_{m.1}} > M$ **then** $l = m + 1$; **else** $r = m - 1$; **end if**
14: **end while**
15: $high = r$;
16: **if** $isFound = 0$ **then**
17: **for** $j = low$ **to** $high$ **do**
18: $atomicAdd(\&r, 1); S[r].bidA = i; S[r].bidB = j$;
19: **end for**
20: **end if**
21: $i += blockDim.x * gridDim.x$;
22: **end while**
23: **return** a solution **or** S

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

Step 1: Use P_i to perform the first binary search in the list B to find a leftmost block $\overline{B_{low}}$, where $1 \leqslant i, low \leqslant k$. The block index *low* should satisfy the following condition:

$$\begin{cases} \overline{a_{i.1}} + \overline{b_{j.e}} > M, & \text{for } 1 \leqslant j < low; \\ \overline{a_{i.1}} + \overline{b_{j.e}} < M, & \text{for } low \leqslant j \leqslant k. \end{cases} \tag{2}$$

Step 2: Use P_i to perform the second binary search in the list B to find a rightmost block $\overline{B_{high}}$, where $low \leqslant high \leqslant k$. The block index *high* should satisfy the following condition:

$$\begin{cases} \overline{a_{i.e}} + \overline{b_{j.1}} > M, & \text{for } low \leqslant j \leqslant high; \\ \overline{a_{i.e}} + \overline{b_{j.1}} < M, & \text{for } high < j \leqslant k. \end{cases} \tag{3}$$

Step 3: Use P_i to write $high - low + 1$ consecutive block pairs $\left(\overline{A_i}, \overline{B_{low}}\right), \left(\overline{A_i}, \overline{B_{low+1}}\right), \cdots, \left(\overline{A_i}, \overline{B_{high}}\right)$ picked by it into the shared memory. After these two binary searches are performed, in order for all picked block pairs to be evenly assigned to each thread in the search stage, the block indices of those picked block pairs need to be written into a vector S.

In our improved pruning stage (IPS), each binary search takes $O(\log k)$ time. If the number of block pairs picked by each thread does not exceed $\log k$, the pruning routine takes $O(3 \log k)$ time; otherwise, it takes $O(2 \log k + k)$ time in the worst case. Indeed, for the k threads, it can be observed that most threads pick at most $\log k$ block pairs in the actual experiment. Even if the number of block pairs picked by a certain thread exceeds $\log k$, it is still far less than k. Therefore, the improved pruning algorithm produces significantly better performance than the original pruning algorithm whose time complexity is $O(k)$.

4.3. The improved search stage

To improve the performance of the search stage, on the basis of Lemmas 1 and 2, we introduce the following Lemmas 5–8 to further shrink the search space of each thread. As the proofs of these four lemmas are similar to the proofs of Lemmas 1 and 2, here, we omit them.

Lemma 5
For any block pair $\left(\overline{A_s}, \overline{B_t}\right)$, where $1 \leqslant s, t \leqslant k$, $\overline{A_s} = \left(\overline{a_{s.1}}, \overline{a_{s.2}}, \cdots, \overline{a_{s.low_A}}, \cdots, \overline{a_{s.e}}\right)$ and $\overline{B_t} = \left(\overline{b_{t.1}}, \overline{b_{t.2}}, \cdots, \overline{b_{t.e}}\right)$, if $\overline{a_{s.low_A - 1}} + \overline{b_{t.1}} < M$ and $\overline{a_{s.low_A}} + \overline{b_{t.1}} > M$, then any element pair $\left(\overline{a_{s.u}}, \overline{b_{t.y}}\right)$ is not a solution of SSP, where $1 < low_A \leqslant e$, $1 \leqslant u < low_A$ and $1 \leqslant y \leqslant e$.

Lemma 6
For any block pair $\left(\overline{A_s}, \overline{B_t}\right)$, where $1 \leqslant s, t \leqslant k$, $\overline{A_s} = \left(\overline{a_{s.1}}, \overline{a_{s.2}}, \cdots, \overline{a_{s.high_A}}, \cdots, \overline{a_{s.e}}\right)$ and $\overline{B_t} = \left(\overline{b_{t.1}}, \overline{b_{t.2}}, \cdots, \overline{b_{t.e}}\right)$, if $\overline{a_{s.high_A}} + \overline{b_{t.e}} < M$ and $\overline{a_{s.high_A + 1}} + \overline{b_{t.e}} > M$, then any element pair $\left(\overline{a_{s.u}}, \overline{b_{t.y}}\right)$ is not a solution of SSP, where $1 \leqslant high_A < e$, $high_A < u \leqslant e$ and $1 \leqslant y \leqslant e$.

Lemma 7
For any block pair $\left(\overline{A_s}, \overline{B_t}\right)$, where $1 \leqslant s, t \leqslant k$, $\overline{A_s} = \left(\overline{a_{s.1}}, \overline{a_{s.2}}, \cdots, \overline{a_{s.e}}\right)$ and $\overline{B_t} = \left(\overline{b_{t.1}}, \overline{b_{t.2}}, \cdots, \overline{b_{t.low_B}}, \cdots, \overline{b_{t.e}}\right)$, if $\overline{a_{s.1}} + \overline{b_{t.low_B - 1}} > M$ and $\overline{a_{s.1}} + \overline{b_{t.low_B}} < M$, then any element pair $\left(\overline{a_{s.x}}, \overline{b_{t.v}}\right)$ is not a solution of SSP, where $1 < low_B \leqslant e$, $1 \leqslant v < low_B$ and $1 \leqslant x \leqslant e$.

Lemma 8
For any block pair $\left(\overline{A_s}, \overline{B_t}\right)$, where $1 \leqslant s, t \leqslant k$, $\overline{A_s} = \left(\overline{a_{s.1}}, \overline{a_{s.2}}, \cdots, \overline{a_{s.e}}\right)$ and $\overline{B_t} = \left(\overline{b_{t.1}}, \overline{b_{t.2}}, \cdots, \overline{b_{t.high_B}}, \cdots, \overline{b_{t.e}}\right)$, if $\overline{a_{s.e}} + \overline{b_{t.high_B}} > M$ and $\overline{a_{s.e}} + \overline{b_{t.high_B + 1}} < M$, then any

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

Algorithm 12 The improved search kernel

Require: $A, B, M, S, r, isFound = 0, e = N/k$

1: $i = blockIdx.x * blockDim.x + threadIdx.x + 1;$
2: **while** $i \leqslant r$ **do**
3: $s = S[i].bidA; t = S[i].bidB; l_A = 1; r_A = e;$
4: **while** $l_A \leqslant r_A$ **and** $isFound = 0$ **do**
5: $m = l_A + (r_A - l_A)/2;$
6: **if** $\overline{a_{s.m}} + \overline{b_{t.1}} = M$ **then** $atomicExch(\&isFound, 1)$; break;
7: **else if** $\overline{a_{s.m}} + \overline{b_{t.1}} > M$ **then** $r_A = m - 1$; **else** $l_A = m + 1$; **end if**
8: **end while**
9: $low_A = l_A; r_A = e;$
10: **while** $l_A \leqslant r_A$ **and** $isFound = 0$ **do**
11: $m = l_A + (r_A - l_A)/2;$
12: **if** $\overline{a_{s.m}} + \overline{b_{t.e}} = M$ **then** $atomicExch(\&isFound, 1)$; break;
13: **else if** $\overline{a_{s.m}} + \overline{b_{t.e}} < M$ **then** $l_A = m + 1$; **else** $r_A = m - 1$; **end if**
14: **end while**
15: $high_A = r_A; l_B = 1; r_B = e;$
16: **while** $l_B \leqslant r_B$ **and** $isFound = 0$ **do**
17: $m = l_B + (r_B - l_B)/2;$
18: **if** $\overline{a_{s.low_A}} + \overline{b_{t.m}} = M$ **then** $atomicExch(\&isFound, 1)$; break;
19: **else if** $\overline{a_{s.low_A}} + \overline{b_{t.m}} < M$ **then** $r_B = m - 1$; **else** $l_B = m + 1$; **end if**
20: **end while**
21: $low_B = l_B; r_B = e;$
22: **while** $l_B \leqslant r_B$ **and** $isFound = 0$ **do**
23: $m = l_B + (r_B - l_B)/2;$
24: **if** $\overline{a_{s.high_A}} + \overline{b_{t.m}} = M$ **then** $atomicExch(\&isFound, 1)$; break;
25: **else if** $\overline{a_{s.high_A}} + \overline{b_{t.m}} > M$ **then** $l_B = m + 1$; **else** $r_B = m - 1$; **end if**
26: **end while**
27: $high_B = r_B; x = low_A; y = low_B;$
28: **while** $x \leqslant high_A$ **and** $y \leqslant high_B$ **and** $isFound = 0$ **do**
29: **if** $\overline{a_{s.x}} + \overline{b_{t.y}} = M$ **then** $atomicExch(\&isFound, 1)$; break; ▷ a solution is found
30: **else if** $\overline{a_{s.x}} + \overline{b_{t.y}} < M$ **then** $x = x + 1$; **else** $y = y + 1$; **end if**
31: **end while**
32: $i += blockDim.x * gridDim.x;$
33: **end while**
34: **return** a solution **or** NULL

element pair $\left(\overline{a_{s.x}}, \overline{b_{t.v}}\right)$ is not a solution of SSP, where $1 \leqslant high_B < e$, $high_B < v \leqslant e$ and $1 \leqslant x \leqslant e$.

Suppose that the block pair $\left(\overline{A_s}, \overline{B_t}\right)$ is assigned to the GPU thread P_i, where $1 \leqslant i, s, t \leqslant k$. On the basis of Lemmas 5 and 6, we use P_i to perform two binary searches in block $\overline{A_s}$ to find the element indices low_A and $high_A$, so as to identify $high_A - low_A + 1$ necessary search elements $\overline{a_{s.low_A}}, \overline{a_{s.low_A+1}}, \cdots, \overline{a_{s.high_A}}$. On the basis of Lemmas 7 and 8, we use P_i to perform two binary searches in block $\overline{B_t}$ to find the element indices low_B and $high_B$, so as to identify $high_B - low_B + 1$ necessary search elements $\overline{b_{t.low_B}}, \overline{b_{t.low_B+1}}, \cdots, \overline{b_{t.high_B}}$. The improved search kernel is shown in Algorithm 12, which consists of the following five steps:

Step 1: Use P_i to perform the first binary search in block $\overline{A_s}$ to find a leftmost element $\overline{a_{s.low_A}}$, where $1 \leqslant low_A \leqslant e$. The element index low_A should satisfy the following condition:

$$\begin{cases} \overline{a_{s.u}} + \overline{b_{t.1}} < M, & \text{for } 1 \leqslant u < low_A; \\ \overline{a_{s.u}} + \overline{b_{t.1}} > M, & \text{for } low_A \leqslant u \leqslant e. \end{cases} \tag{4}$$

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

Step 2: Use P_i to perform the second binary search in block $\overline{A_s}$ to find a rightmost element $\overline{a_{s.high_A}}$, where $low_A \leqslant high_A \leqslant e$. The element index $high_A$ should satisfy the following condition:

$$\begin{cases} \overline{a_{s.u}} + \overline{b_{t.e}} < M, & \text{for } low_A \leqslant u \leqslant high_A; \\ \overline{a_{s.u}} + \overline{b_{t.e}} > M, & \text{for } high_A < u \leqslant e. \end{cases} \tag{5}$$

Step 3: Use P_i to perform the first binary search in block $\overline{B_t}$ to find a leftmost element $\overline{b_{t.low_B}}$, where $1 \leqslant low_B \leqslant e$. The element index low_B should satisfy the following condition:

$$\begin{cases} \overline{a_{s.low_A}} + \overline{b_{t.v}} > M, & \text{for } 1 \leqslant v < low_B; \\ \overline{a_{s.low_A}} + \overline{b_{t.v}} < M, & \text{for } low_B \leqslant v \leqslant e. \end{cases} \tag{6}$$

Step 4: Use P_i to perform the second binary search in block $\overline{B_t}$ to find a rightmost element $\overline{b_{t.high_B}}$, where $low_B \leqslant high_B \leqslant e$. The element index $high_B$ should satisfy the following condition:

$$\begin{cases} \overline{a_{s.high_A}} + \overline{b_{t.v}} > M, & \text{for } low_B \leqslant v \leqslant high_B; \\ \overline{a_{s.high_A}} + \overline{b_{t.v}} < M, & \text{for } high_B < v \leqslant e. \end{cases} \tag{7}$$

Step 5: Use P_i to perform the search routine of Horowitz and Sahni's $two\text{-}list$ algorithm on the block pair $(\overline{A_s}, \overline{B_t})$, where $\overline{A_s} = \left[\overline{a_{s.low_A}}, \overline{a_{s.low_A+1}}, \cdots, \overline{a_{s.high_A}}\right]$ and $\overline{B_t} = \left[\overline{b_{t.low_B}}, \overline{b_{t.low_B+1}}, \cdots, \overline{b_{t.high_B}}\right]$, after these four binary searches have completed.

It is readily seen that the time complexity of the improved search stage (ISS) is $O(4\log(N/k) + 2N/k)$ in the worst case, which is clearly bounded by $O(N/k)$. Although the worst-case time complexity of the original search stage (OSS) is also $O(N/k)$, however, in most cases, we observe that the search space of each thread can be greatly reduced after these four binary searches are performed in the ISS. Therefore, the ISS might significantly outperform the OSS.

5. EXPERIMENTAL EVALUATION

In this section, we first present the experimental setup. Then, we evaluate the effectiveness of the proposed GPU implementation. Finally, we analyze the performance of the improved parallel *two-list* algorithm.

5.1. Experimental setup

In this paper, apart from the proposed GPU implementation, we implement the parallel *two-list* algorithm on multicore CPUs using OpenMP, so as to compare the performance of the GPU version with the multicore CPU version. Furthermore, to show the performance of the two parallel implementations, we compare their performance with the two best known sequential algorithms. One is Horowitz and Sahni's *two-list* algorithm [6], which solves SSP in $O(n2^{n/2})$ time with $O(2^{n/2})$ memory space. The other is the dynamic programming algorithm [7], which solves SSP in $O(nM)$ time and memory space.

To check the effectiveness and scalability of our approach, a series of experiments are carried out on a dual quad-core 2.3 GHz AMD Opteron 2376 machine with 4 GB main memory, and the following two different NVIDIA graphics cards are used:

- The GTX 465 GPU has 352 CUDA cores, 607 MHz processor clock, 1 GB GDDR5 RAM, 102.6 GB/s memory bandwidth, and 802 MHz memory clock.
- The Tesla C2050 GPU has 448 CUDA cores, 1.15 GHz processor clock, 3 GB GDDR5 RAM, 144 GB/s memory bandwidth, and 1.5 GHz memory clock.

In software, the testing system is built on top of the Linux (Ubuntu 10.10) operating system with NVIDIA CUDA driver version 4.2 and GCC version 4.4.5.

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

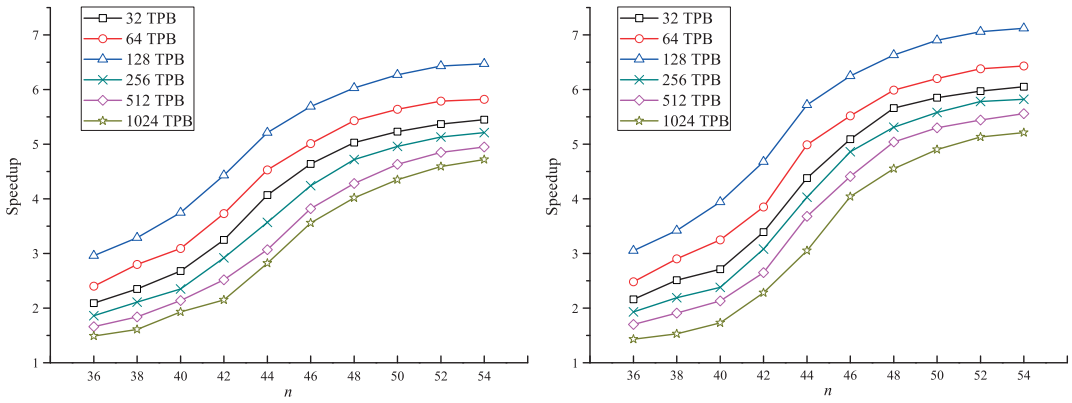


Figure 4. The speedups of GPU implementation on two different GPU cards using six different settings for threads per block.

We conduct our tests on randomly generated instances of SSP, and we use a random number generator to produce the input vectors W whose sizes range from 36 to 54. They present the following features:

- w_i is randomly drawn in $[1, 10^8]$, $i \in \{1, \cdots, n\}$.
- $M = \alpha \sum_{i=1}^{n} w_i$, $\alpha \in \{0.2, \cdots, 0.8\}$.
- $w_i < M$, $i \in \{1, \cdots, n\}$.

In our experiments, we consider the following 10 different problem sizes: 36, 38, 40, 42, 44, 46, 48, 50, 52 and 54. Note that the sequential *two-list* algorithm [6] and the parallel *two-list* algorithm [17] all require $O(2^{n/2})$ memory space, implying that the problem size will be limited by the available memory on the testing system. In addition, considering the real-world applications and the worst-case execution time, it is necessary to take into account different knapsack capacities.

For each problem size, we randomly generate 1000 instances. The average execution time is considered, and the execution time is measured in milliseconds. The speedup is defined as T_s / T_p, where T_s and T_p represent the running time of the sequential implementation and parallel implementation, respectively.

Moreover, to achieve effective and robust performance for the GPU implementation, we should adopt the optimum number of TPB to keep the GPU multiprocessors as active as possible. Here, we conduct our preliminary speedup performance tests on the GTX 465 GPU and the Tesla C2050 GPU using six different settings for TPB (respectively, 32, 64, 128, 256, 512, and 1024). In this experiment, we specify $M = 0.5 \sum_{i=1}^{n} w_i$, and T_s is obtained by executing Horowitz and Sahni's sequential *two-list* algorithm. Note that the maximum number of TPB is 1024 for both cards, thus at most 1024 threads can be specified in each block.

Figure 4 shows the speedups of the GPU implementation on two different GPU cards using six different settings for TPB. For the 10 different problem sizes, among the six different settings for TPB, Figures 4(a) and 4(b) show that the 128 TPB case seems to be the most efficient. It always gives more stable and scalable speedup, because it can obtain the best multiprocessor occupancy. Hence, we choose the 128 TPB case for further performance tests on both the GTX 465 GPU and Tesla C2050 GPU.

5.2. *Comparison with CPU implementation*

To accurately evaluate the performance of the proposed GPU implementation, for each randomly generated instance, we first run the dynamic programming algorithm and Horowitz and Sahni's sequential *two-list* algorithm on the CPU. Then, we run the parallel *two-list* algorithm on multicore

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

Table II. Experimental results on the CPU and two different GPU cards.

		Sequential algorithm		Parallel *two-list* algorithm								
		DP	*two-list*	Multicore CPUs			GTX 465 GPU			Tesla C2050 GPU		
M	n	Time	Time	Time	Sp-DP	Sp-TL	Time	Sp-DP	Sp-TL	Time	Sp-DP	Sp-TL
	36	27.93	61.29	26.51	1.05	2.31	20.72	1.35	2.96	20.09	1.39	3.05
	38	53.15	115.24	47.96	1.11	2.40	35.08	1.52	3.29	33.67	1.58	3.42
Case 1	40	110.54	205.45	82.39	1.34	2.49	54.85	2.02	3.75	52.17	2.12	3.94
	42	230.64	358.29	132.17	1.74	2.71	80.89	2.85	4.43	76.59	3.01	4.68
	44	480.16	645.12	215.58	2.23	2.99	123.78	3.88	5.21	112.88	4.25	5.72
	46	2127.87	1151.77	365.69	5.82	3.15	202.50	10.51	5.69	184.17	11.55	6.25
	48	4183.82	2132.49	650.35	6.43	3.28	353.82	11.82	6.03	321.88	13.00	6.63
Case 2	50	8714.37	4093.46	1217.46	7.16	3.36	652.89	13.35	6.27	593.12	14.69	6.90
	52	16685.87	7971.76	2234.80	7.47	3.57	1239.60	13.46	6.43	1129.92	14.77	7.06
	54	33935.49	16124.25	4377.58	7.75	3.68	2490.87	13.62	6.47	2265.74	14.98	7.12

Case 1: $M = 0.5\sum_{i=1}^{n} w_i \cong \frac{1}{8}2^{n/2}$ Case 2: $M = 0.5\sum_{i=1}^{n} w_i \cong \frac{1}{4}2^{n/2}$
DP, dynamic programming; Sp-DP, speedup over the sequential dynamic programming algorithm; Sp-TL, speedup over the sequential *two-list* algorithm.

CPUs with eight OpenMP threads. Finally, we run the parallel *two-list* algorithm on two different GPU cards. Table II shows the execution times of the dynamic programming algorithm and Horowitz and Sahni's sequential *two-list* algorithm, and it also shows the execution times and speedups of the multicore CPU implementation and the GPU implementation, respectively, for the 10 different problem sizes. In the experiments, the knapsack capacity M is half the total weight, which is moderate for the first five problem sizes and is large for the other five problem sizes.

In Table II, it is not hard to see that the dynamic programming technique is attractive for moderate knapsack capacity, but Horowitz and Sahni's *two-list* algorithm is attractive for large knapsack capacity. It can be found that when the problem size increases, the execution time and the speedup factor also increase, and the multicore CPU implementation and the GPU implementation show better performance over the two sequential implementations.

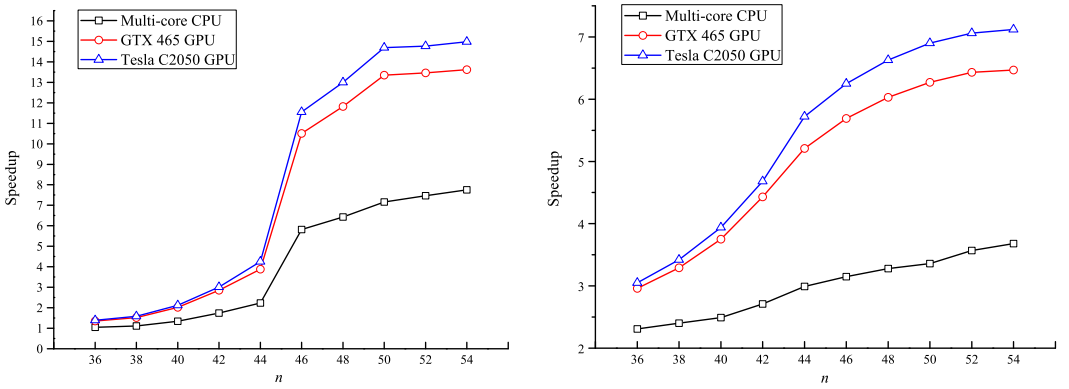
The performance comparison between the GPU implementation and the two sequential implementations shows that the speedup is not substantial for small problem sizes ($n \leqslant 42$). This is associated with the fact that there may not be enough work to fully utilize the available hardware parallelism of the GPU. On the other hand, the execution of the GPU program involves some fixed overheads: thread creation and destruction overhead, GPU initialization overhead, GPU memory allocation overhead, GPU synchronization overhead, and data transfer overhead. This makes GPU not suitable for small computational tasks, as these fixed overheads are higher than the kernel execution time. However, for large problem sizes ($n > 42$), the GPU implementation can achieve substantial speedup, in comparison with the two sequential implementations, it obtains 14.98 and 7.12 speedup, respectively, when $n = 54$. Hence, the proposed GPU implementation is most fit for large-scale SSP.

Figures 5(a) and 5(b) illustrate the speedups of the multicore CPU implementation and the GPU implementation over the dynamic programming algorithm and the sequential *two-list* algorithm. As we can see, as long as the problem size increases, the speedup factor grows accordingly. It is obvious that the GPU implementation has much better performance than the multicore CPU implementation, because a large number of cores are available on both the GTX 465 GPU and Tesla C2050 GPU. Note that the larger the problem size is, the slower the speedup increases, and the speedup will gradually reach a peak.

The results from Figures 5(a) and 5(b) also show that the GTX 465 GPU and the Tesla C2050 GPU offer similar performance for small problem sizes ($n \leqslant 42$). However, when the problem size exceeds 42, the Tesla C2050 GPU yields higher speedup than the GTX 465 GPU. This is due to the fact that the Tesla C2050 GPU enjoys more available streaming processor cores and higher memory bandwidth than the GTX 465 GPU.

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe



(a) The speedups over the dynamic programming algorithm (b) The speedups over the sequential *two-list* algorithm

Figure 5. The speedups obtained using multicore CPUs and two different GPU cards.

Table III. The execution time of each stage of the *two-list* algorithm.

	Sequential *two-list* algorithm			Parallel *two-list* algorithm			
n	Generation time	Search time	Total time	Generation time	Pruning time	Search time	Total time
36	43.10	18.19	61.29	18.49	0.55	1.06	20.09
38	80.80	34.44	115.24	31.10	0.73	1.84	33.67
40	146.02	59.44	205.45	48.07	1.08	3.01	52.17
42	253.57	104.72	358.29	70.05	2.13	4.40	76.59
44	457.69	187.43	645.12	102.01	4.28	6.59	112.88
46	801.62	350.15	1151.77	171.50	5.01	7.67	184.17
48	1450.02	682.47	2132.49	300.31	8.63	12.95	321.88
50	2845.61	1247.85	4093.46	551.16	16.04	25.92	593.12
52	5575.43	2396.33	7971.76	1052.71	28.02	49.19	1129.92
54	11133.43	4990.82	16124.25	2118.37	57.25	90.12	2265.74

5.3. Analysis of the execution time

Table III shows the execution time of each stage of the sequential *two-list* algorithm and the parallel *two-list* algorithm. Note that the parallel *two-list* algorithm is executed on the Tesla C2050 GPU. According to Table III, for the sequential *two-list* algorithm, the generation stage and the search stage consume 68.00–71.07% and 28.93–32.00% of the total execution time, respectively, for the 10 different problem sizes. In contrast, for the parallel *two-list* algorithm, the most time-consuming process is the generation stage, which takes 90.37–93.50% of the total execution time, whereas the pruning stage and the search stage consume 2.07–3.79% and 3.98–5.84% of the total execution time, respectively. The performance comparison of the parallel implementation with the sequential one shows that the parallel generation stage obtains good efficiency, and the parallel search stage achieves better efficiency.

In addition, it is worth mentioning that the more GPU threads are used, the longer the execution time of the pruning stage is, and the shorter the execution time of the search stage is. Therefore, to obtain the best performance, it is necessary to adopt the optimum number of GPU threads during the pruning and search stages, provided that it does not exceed the maximum number of available GPU threads. In our experiments, the number of GPU threads used is $k = 2^{n/4}$.

5.4. Analysis of data transfer

It is generally appreciated that the data transfer between CPU and GPU can significantly affect the performance of GPU applications. In this section, we intend to do an analysis of the percentage

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

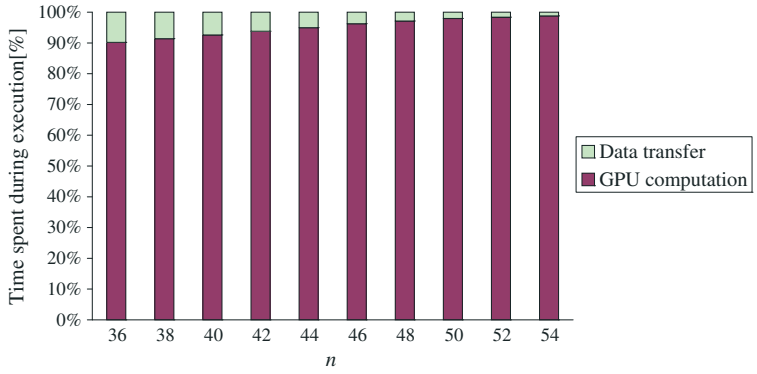


Figure 6. The percentage of the time spent by GPU computation and data transfer.

of the time spent by GPU computation and data transfer for different problem sizes, as shown in Figure 6. These measurements are conducted on the Tesla C2050 GPU and the knapsack capacity is half the total weight.

The total execution time of our GPU implementation mainly consists of the GPU computation time and the data transfer time. The GPU computation mainly includes the following: kernel execution, thread creation and destruction, GPU initialization, GPU memory allocation, and GPU synchronization. The data transfer mainly includes the following: transferring W_1, W_2 and M to the GPU in the beginning of the generation stage; copying the current iteration number to the GPU during the execution of the generation stage; transferring the solution back to the CPU after the pruning stage or the search phase has completed.

Figure 6 shows that the GPU computation dominates the overwhelming majority of the overall execution time. Specifically, with the increase of problem size, the data transfer time grows slightly, but the GPU computation time grows greatly, meaning that the percentage of the time spent by data transfer will be reduced progressively. The results from Figure 6 reveal that data transfer between CPU and GPU is not a performance bottleneck for our GPU implementation. The reasons are as follows: we execute the entire algorithm on the GPU; we directly read data from the lists A and B and write data to them on the device side; we use the vector-based iterative implementation mechanism during the partition process, which can help us to avoid transferring large amount of partition information between CPU and GPU.

5.5. Analysis of the improved generation stage

Before comparing the performance of the IGS with that of the OGS, it is necessary to study how the knapsack capacity affects the number of subset sums that can be discarded in the list A. Here, we consider eight different knapsack capacities as follows: $M = \alpha \sum_{i=1}^{n} w_i$, $\alpha \in \{0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5, 0.55\}$. Each randomly generated instance needs to be performed eight times according to the eight different knapsack capacities.

Let DS_{total} denote the total number of subset sums to be discarded in the list A, and λ denote the discard rate. Note that the total number of subset sums in the list A without performing any discard is N, so the discard rate can be calculated as $\lambda = DS_{total} / N$.

Table IV shows the discard rate achieved by using eight different knapsack capacities for each problem size. It is easy to find that the smaller the knapsack capacity, the higher the discard rate, and vice versa. For example, when $M = 0.2 \sum w_i$, the average discard rate of 10 problem sizes is 97.31%, however, when $M = 0.55 \sum w_i$, the average discard rate drops to only 1.63%.

Table V shows the execution time of the OGS and the IGS on the Tesla C2050 GPU. For the OGS, the knapsack capacity does not affect its execution time, therefore its execution time are the same for eight different knapsack capacities. For the IGS, it is obvious that the smaller the knapsack capacity, the shorter its execution time. For example, when the knapsack capacities are $0.2\sum w_i$, $0.3\sum w_i$, $0.4\sum w_i$ and $0.5\sum w_i$, the execution times of the OGS are reduced by about 43%, 32%,

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

Table IV. The discard rate for different knapsack capacities.

n \ M	$0.2\sum w_i$	$0.25\sum w_i$	$0.3\sum w_i$	$0.35\sum w_i$	$0.4\sum w_i$	$0.45\sum w_i$	$0.5\sum w_i$	$0.55\sum w_i$
36	0.9722	0.9213	0.8124	0.6057	0.3722	0.1818	0.0731	0.0160
38	0.9755	0.9232	0.8110	0.6061	0.3749	0.1858	0.0729	0.0159
40	0.9719	0.9231	0.8111	0.6072	0.3728	0.1840	0.0712	0.0162
42	0.9709	0.9203	0.8132	0.6010	0.3758	0.1858	0.0716	0.0163
44	0.9733	0.9226	0.8107	0.6079	0.3714	0.1869	0.0726	0.0161
46	0.9750	0.9215	0.8113	0.6046	0.3702	0.1826	0.0743	0.0164
48	0.9768	0.9253	0.8103	0.6037	0.3732	0.1802	0.0728	0.0163
50	0.9719	0.9205	0.8143	0.6054	0.3747	0.1866	0.0733	0.0168
52	0.9728	0.9225	0.8139	0.6078	0.3711	0.1851	0.0729	0.0161
54	0.9710	0.9235	0.8102	0.6081	0.3706	0.1833	0.0744	0.0167
Average	0.9731	0.9224	0.8118	0.6057	0.3727	0.1842	0.0729	0.0163

Table V. Execution time comparison between IGS and OGS on the Tesla C2050 GPU.

		IGS							
n	OGS	$0.2\sum w_i$	$0.25\sum w_i$	$0.3\sum w_i$	$0.35\sum w_i$	$0.4\sum w_i$	$0.45\sum w_i$	$0.5\sum w_i$	$0.55\sum w_i$
36	18.49	10.57	11.52	12.65	13.92	15.84	17.34	18.14	18.39
38	31.10	17.79	19.38	21.27	23.42	26.65	29.17	30.51	30.95
40	48.07	27.50	29.95	32.88	36.20	41.19	45.09	47.16	47.83
42	70.05	40.07	43.64	47.92	52.75	60.04	65.71	68.72	69.70
44	102.01	58.35	63.55	69.77	76.81	87.42	95.68	100.07	101.50
46	171.50	98.10	106.84	117.30	129.14	146.97	160.87	168.24	170.64
48	300.31	171.78	187.09	205.41	226.13	257.36	281.69	294.60	298.81
50	551.16	315.26	343.37	376.99	415.02	472.34	516.99	540.69	548.40
52	1052.71	602.15	655.84	720.05	792.69	902.17	987.44	1032.71	1047.44
54	2118.37	1211.71	1319.74	1448.96	1595.13	1815.44	1987.03	2078.12	2107.78

IGS, improved generation stage; OGS, original generation stage.

14% and 2%, respectively, for each problem size. The execution time comparison between IGS and OGS shows that our IGS yields significant performance benefits when $M < 0.5\sum w_i$.

5.6. Analysis of the improved pruning stage

Because the number of block pairs picked by each GPU thread can affect the performance of the IPS, this section first shows the excess rate and then presents a performance comparison between the IPS and the original pruning stage (OPS).

Let EX_{total} denote the total number of the threads, which pick more than $\log k$ block pairs, and γ denote the excess rate. Note that the total number of threads used in the IPS is k, the excess rate can be calculated as $\gamma = EX_{total}/k$.

Table VI shows the excess rate achieved by using seven different knapsack capacities for each problem size. We can easily see that most threads pick less than $\log k$ block pairs, especially when $M = 0.5\sum w_i$, almost all threads pick less than $\log k$ block pairs. The results from VI reveal that our IPS should be able to achieve better performance.

Table VII shows the execution time of the OPS and the IPS on the Tesla C2050 GPU for $M = 0.5\sum w_i$. It is obvious that our IPS achieves significantly better performance than the OPS. For example, when the problem sizes are 36, 42, 48, and 54, the execution times of the OPS are reduced by 80%, 84.98%, 88.53%, and 91.41%, respectively.

5.7. Analysis of the improved search stage

Because the performance of the OSS is improved by reducing the search spaces of those picked block pairs, this section first shows the search space reduction rate and then presents a performance comparison between the ISS and the OSS.

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

Table VI. The excess rate for different knapsack capacities.

n \ M	$0.2\sum w_i$	$0.3\sum w_i$	$0.4\sum w_i$	$0.5\sum w_i$	$0.6\sum w_i$	$0.7\sum w_i$	$0.8\sum w_i$
36	0.0031	0.0141	0.0154	0.0002	0.0155	0.0142	0.0032
38	0.0025	0.0136	0.0149	0.0001	0.0150	0.0137	0.0026
40	0.0021	0.0116	0.0136	0.0001	0.0137	0.0115	0.0021
42	0.0019	0.0113	0.0133	0.0000	0.0134	0.0112	0.0018
44	0.0016	0.0110	0.0131	0.0000	0.0131	0.0110	0.0015
46	0.0014	0.0098	0.0129	0.0000	0.0130	0.0098	0.0013
48	0.0012	0.0096	0.0126	0.0000	0.0127	0.0095	0.0011
50	0.0010	0.0094	0.0123	0.0000	0.0124	0.0093	0.0010
52	0.0009	0.0092	0.0120	0.0000	0.0121	0.0091	0.0008
54	0.0008	0.0090	0.0117	0.0000	0.0118	0.0089	0.0007
Average	0.0017	0.0109	0.0132	0.0000	0.0133	0.0108	0.0016

Table VII. Execution time comparison between IPS and OPS for $M = 0.5\sum w_i$.

	$n=36$	$n=38$	$n=40$	$n=42$	$n=44$	$n=46$	$n=48$	$n=50$	$n=52$	$n=54$
OPS	0.55	0.73	1.08	2.13	4.28	5.01	8.63	16.04	28.02	57.25
IPS	0.11	0.14	0.17	0.32	0.59	0.65	0.99	1.72	2.67	4.92

OPS, original pruning stage; IPS, improved pruning stage.

Table VIII. The search space reduction rate for different knapsack capacities.

	$0.3\sum w_i$		$0.4\sum w_i$		$0.5\sum w_i$		$0.6\sum w_i$		$0.7\sum w_i$	
n	A	B	A	B	A	B	A	B	A	B
36	0.4430	0.5119	0.4554	0.4966	0.4522	0.4999	0.4563	0.4973	0.4429	0.5102
38	0.4342	0.5160	0.4520	0.4988	0.4515	0.5082	0.4543	0.4989	0.4335	0.5191
40	0.4353	0.5108	0.4572	0.4956	0.4518	0.5015	0.4572	0.4996	0.4448	0.5183
42	0.4419	0.5103	0.4596	0.4977	0.4521	0.4954	0.4562	0.4982	0.4509	0.5155
44	0.4443	0.5173	0.4580	0.4981	0.4512	0.5016	0.4577	0.4968	0.4436	0.5182
46	0.4563	0.5173	0.4568	0.4951	0.4515	0.5015	0.4542	0.4992	0.4559	0.5191
48	0.4398	0.5198	0.4567	0.4997	0.4516	0.5013	0.4567	0.4996	0.4394	0.5124
50	0.4393	0.5190	0.4571	0.4953	0.4517	0.5016	0.4571	0.4953	0.4393	0.5190
52	0.4405	0.5153	0.4564	0.4951	0.4519	0.5052	0.4564	0.4961	0.4341	0.5139
54	0.4418	0.5156	0.4570	0.4970	0.4520	0.5024	0.4565	0.4945	0.4419	0.5175
Average	0.4416	0.5153	0.4566	0.4969	0.4517	0.5019	0.4562	0.4976	0.4426	0.5163

Suppose that the block pair $(\overline{A_s}, \overline{B_t})$ is assigned to the thread P_i, where $1 \leqslant i, s, t \leqslant k$. For the OSS, the search space of block $\overline{A_s}$ ranges from $\overline{a_{s.1}}$ to $\overline{a_{s.e}}$, and the search space of block $\overline{B_t}$ ranges from $\overline{b_{t.1}}$ to $\overline{b_{t.e}}$. For the ISS, the search space of block $\overline{A_s}$ ranges from $\overline{a_{s.low_A}}$ to $\overline{a_{s.high_A}}$, and the search space of block $\overline{B_t}$ ranges from $\overline{b_{t.low_B}}$ to $\overline{b_{t.high_B}}$. Therefore, the search space reduction rates of block $\overline{A_s}$ and block $\overline{B_t}$ are $(e-(high_A - low_A + 1))/e$ and $(e-(high_B - low_B + 1))/e$, respectively.

Table VIII shows the average search space reduction rate of all picked block pairs achieved by using five different knapsack capacities for each problem size. Columns 'A' and 'B' represent the average search space reduction rate of these blocks from lists A and B in all picked block pairs, respectively. It is clear that the search space of each picked block pair is greatly reduced; the search space reduction rate of one block from lists A and B in each picked block pair are around 45% and 50%, respectively.

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

Table IX. Execution time comparison between ISS and OSS for $M = 0.5 \sum w_i$.

	$n = 36$	$n = 38$	$n = 40$	$n = 42$	$n = 44$	$n = 46$	$n = 48$	$n = 50$	$n = 52$	$n = 54$
OSS	1.06	1.84	3.01	4.40	6.59	7.67	12.95	25.92	49.19	90.12
ISS	0.4978	0.8616	1.4151	2.0629	3.0525	3.6057	6.1235	12.2059	23.1607	42.3152

OSS, original search stage; ISS, improved search stage.

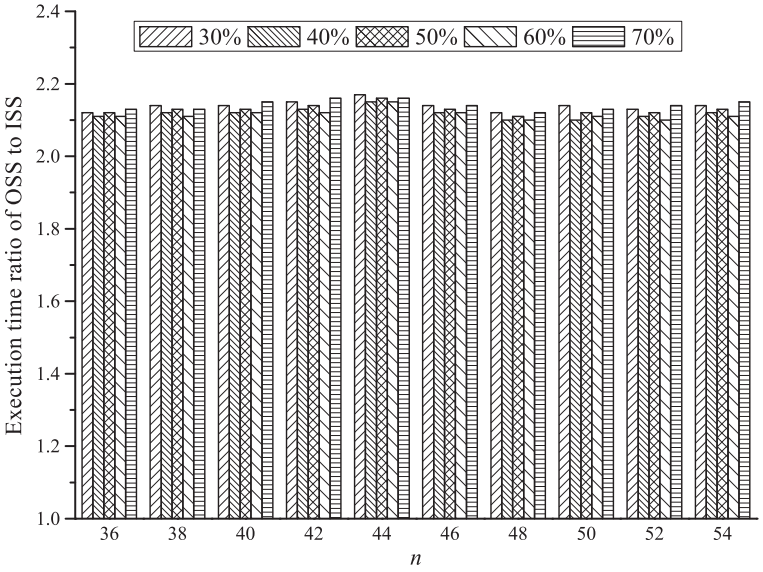


Figure 7. The execution time ratios of original search stage to improved search stage for five different knapsack capacities.

Table IX shows the execution time of the OSS and the ISS on the Tesla C2050 GPU for $M = 0.5 \sum w_i$. We can easily observe that the ISS outperforms the OSS for each problem size, and the execution time of the OSS is reduced by an average of 53.05%.

Because the knapsack capacity can affect the search time, for clarity, the execution time ratios of OSS to ISS for five different knapsack capacities (30%, 40%, 50%, 60%, 70% of total weight) are shown in Figure 7. The execution time ratio is around 2.13 for each problem size, and the results show that the ISS gives a significant performance benefit.

6. CONCLUSIONS

In this paper, on the basis of the parallel *two-list* algorithm of Li *et al*., we propose and evaluate an original parallel implementation of the algorithm for solving SSP on a GPU using CUDA. Apart from studying how to implement the three stages of the algorithm by exploiting fine-grained data parallelism and thread parallelism offered by the GPU, a great deal of programming effort has been devoted to achieving the best performance. Reasonable task distribution between CPU and GPU is essential to improve the performance. The vast majority of computational tasks are performed on the GPU, so we do not have to transfer large amounts of data back and forth between CPU and GPU, which greatly reduces the communication cost between CPU and GPU. Efficient memory management is necessary to reduce memory access latency. On the basis of multilevel GPU memory hierarchies, we select the best memory space to store the data required for running the algorithm. To convert the recursive solution into an iterative one during the generation stage, we develop a new vector-based iterative implementation mechanism instead of the explicit recursion. Furthermore, a significant performance benefit comes from the improved parallel *two-list* algorithm, including the IGS, the IPS, and the ISS.

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

A series of experiments are conducted to test the performance of our GPU implementation under different experimental environments. The experimental results show that the GPU implementation has much better performance than the CPU implementation and can achieve significant speedup on different GPU cards. The experimental results also show that the improved algorithm can bring significant performance benefits for our GPU implementation.

From a hardware point of view, the major consideration for our GPU implementation is the memory available on the GPU. Up to now, the largest single GPU memory space available does not exceed 8 GB, which is far from meeting the memory requirements for large-scale SSP. In future work, we will explore new techniques to solve large-scale SSP on heterogeneous CPU/GPU cluster systems with less communication overhead and optimal load balancing.

ACKNOWLEDGEMENTS

The authors would like to thank the editor and anonymous reviewers for their valuable suggestions which considerably helped us to enhance this paper. This work was partially funded by the Key Program of National Natural Science Foundation of China (Grant No. 61133005), and the National Natural Science Foundation of China (Grant Nos. 61070057, 61370095, 61173013, 61370098)
NSFC61070057, 61370095, 61173013, 61370098
Key Program of NSFC61133005

REFERENCES

1. Gary MR, Johnson DS. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. WH Freeman and Company: New York, 1979.
2. Dietrich BL, Escudero LF, Chance F. Efficient reformulation for 0–1 programs–methods and computational results. *Discrete Applied Mathematics* 1993; **42**(2):147–175.
3. Dyckhoff H. A new linear programming approach to the cutting stock problem. *Operations Research* 1981; **29**(6):1092–1104.
4. Kleywegt AJ, Papastavrou JD. The dynamic and stochastic knapsack problem with random sized items. *Operations Research* 2001; **49**(1):26–41.
5. Martello S, Toth P. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Inc.: New York, NY, 1990.
6. Horowitz E, Sahni S. Computing partitions with applications to the knapsack problem. *Journal of the ACM (JACM)* 1974; **21**(2):277–292.
7. Bellman RE. *Dynamic Programming*. Princeton University Press: Princeton, New Jersey, 1957.
8. Schroeppel R, Shamir A. A T=$O(2^{n/2})$, S=$O(2^{n/4})$ algorithm for certain NP-complete problems. *SIAM Journal on Computing* 1981; **10**(3):456–464.
9. Karnin ED. A parallel algorithm for the knapsack problem. *IEEE Transactions on Computers* 1984; **100**(5):404–408.
10. Ferreira AG. A parallel time/hardware tradeoff $T \cdot H = O(2^{n/2})$ for the knapsack problem. *IEEE Transactions on Computers* 1991; **40**(2):221–225.
11. Chang HKC, Chen JJR, Shyu SJ. A parallel algorithm for the knapsack problem using a generation and searching technique. *Parallel Computing* 1994; **20**(2):233–243.
12. Lou DC, Chang CC. A parallel two-list algorithm for the knapsack problem. *Parallel Computing* 1997; **22**(14):1985–1996.
13. Sanches CAA, Soma NY, Yanasse HH. Comments on parallel algorithms for the knapsack problem. *Parallel Computing* 2002; **28**(10):1501–1505.
14. 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.
15. Chedid FB. An optimal parallelization of the two-list algorithm of cost $O(2^{n/2})$. *Parallel Computing* 2008; **34**(1):63–65.
16. Chedid FB. A note on developing optimal and scalable parallel two-list algorithms. In *Algorithms and Architectures for Parallel Processing*. Springer-Verlag: Berlin, 2012; 148–155.
17. 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.
18. Ryoo S, Rodrigues CI, Stone SS, Stratton JA, Ueng SZ, Baghsorkhi SS, Hwu WmW. Program optimization carving for GPU computing. *Journal of Parallel and Distributed Computing* 2008; **68**(10):1389–1401.
19. Che S, Boyer M, Meng J, 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.
20. Bokhari SS. Parallel solution of the subset-sum problem: an empirical study. *Concurrency and Computation: Practice & Experience* 2012; **24**(18):2241–2254.

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

21. Boyer V, El Baz D, Elkihel M. Solving knapsack problems on GPU. *Computers & Operations Research* 2012; **39**(1):42–47.
22. Lalami ME, El-Baz D. GPU implementation of the Branch and Bound method for knapsack problems. *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW),* IEEE, Shanghai, China, 2012; 1769–1777.
23. 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, Czech Republic, 2010; 64–70.
24. Akl SG, Santoro N. Optimal parallel merging and sorting without memory conflicts. *IEEE Transactions on Computers* 1987; **100**(11):1367–1369.

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe