A Novel CPU-GPU Cooperative Implementation of A Parallel Two-List Algorithm for the Subset-Sum Problem

Jing Liu

College of Information Science and Engineering,

Hunan University

Feb. 15, 2014

Outline

- Background
- The CPU-GPU Cooperative Computing Environment
- The CPU-GPU Cooperative Implementation
- Experimental Evaluation
- Conclusions

Background

1. Introduction of subset-sum problem

■ 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 binary ***n***-tuple solution $X = [x_1, x_2, \cdots, x_n]$ for the equation

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

■ **Hard to solve:** SSP is well-known to be **NP-complete,** and it is a special case of the 0/1 knapsack problem.

■ **Many real-world applications:** stock cutting, cargo loading, capital budgeting, job scheduling, and workload allocation, etc.

Background

2. Related work of solving subset-sum problem

- **Many techniques have been developed to solve SSP**
  - **Exact algorithms:** dynamic programming, branch-and-bound, **two-list algorithm**, etc
  - **Heuristic algorithms:** genetic algorithm, local search, etc

- **Sequential two-list algorithm:** the best known sequential algorithm for exactly solving SSP in **time** $O(n2^{n/2})$ with $O(2^{n/2})$ memory space.

- **Parallel two-list algorithm:** to reduce the computation time of SSP, the parallelization of the two-list algorithm has been extensively discussed in recent years.

Background

2. Related work of solving subset-sum problem

■ Parallel implementation for the subset-sum problem

Recently, **heterogeneous CPU-GPU system** has been widely used, which is a powerful way to deal with time-intensive problems. To solve SSP on **multi-core CPU** or **many-core GPU**, some work has been done.

Table 1. Parallel implementation for the subset-sum problem

	dynamic programming	branch-and-bound	**two-list algorithm**	genetic algorithm
CPU-only implmentation	✓	✓	✓	✓
GPU-only implmentation	✓	✓	✓	✓
CPU-GPU cooperative implmentation	×	×	×	×

As shown in Table 1, **no implementation combines both CPU and GPU** to accelerate solving the subset-sum problem.

Background

2. Related work of solving subset-sum problem

- **The CPU-only, GPU-only implementations fail to fully utilize all the CPU cores and the GPU resources at the same time**

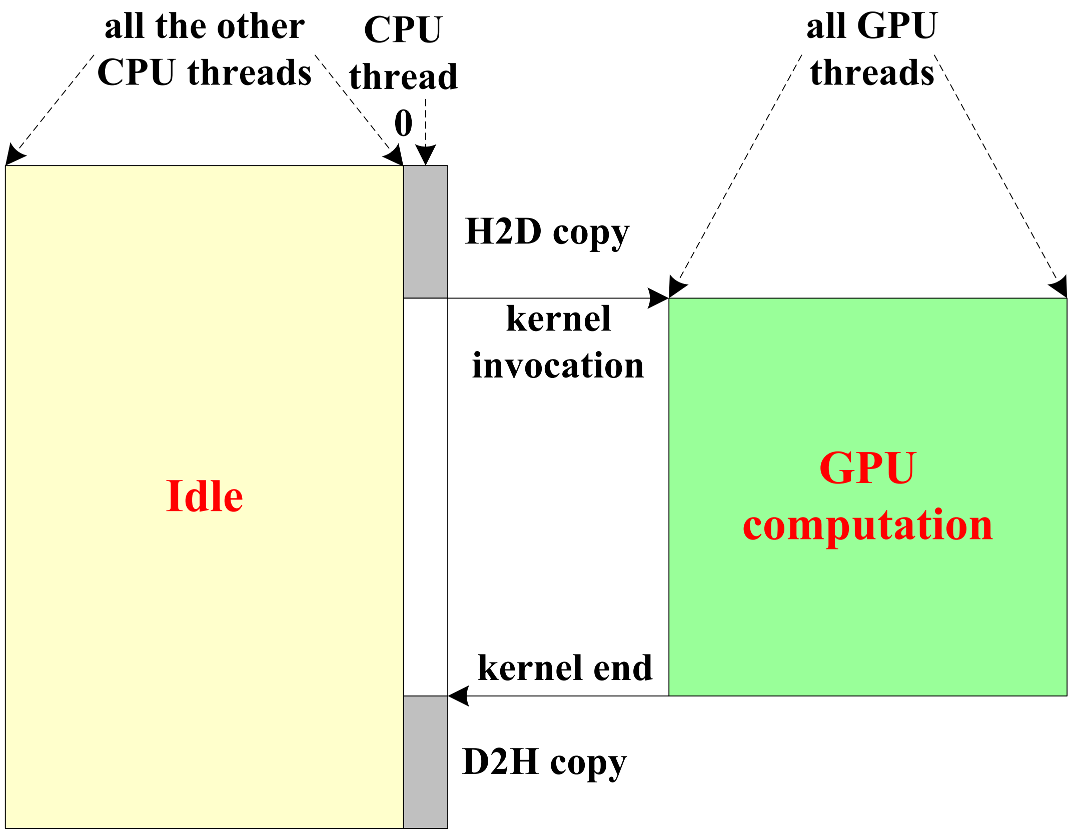


Fig. 1 The GPU-only implementation

In the GPU-only implementation, only one CPU thread is used to control and to communicate with the GPU, all the other CPU threads are in idle state while the GPU performs some tasks.

Problem:

this leads to large amounts of available CPU resources are wasted.

Background

3. Motivation

In order to effectively solve SSP in a heterogeneous system, based on the optimal parallel two-list algorithm [1], we **propose a novel CPU-GPU cooperative implementation of the algorithm,** i.e., **find an effective method to make full use of all the available resources of both CPU and GPU to accelerate solving SSP**.

[1] 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.

Background

4. Key challenges

The parallel two-list algorithm is a typical example of an irregularly structured problem, so it is hard to implement the algorithm on both CPU and GPU.

How to assign the most suitable workload to the CPU and GPU to maximize the utilization of all computational resources.

The CPU-GPU Cooperative Computing Environment

1. The first CPU-GPU cooperative computing method

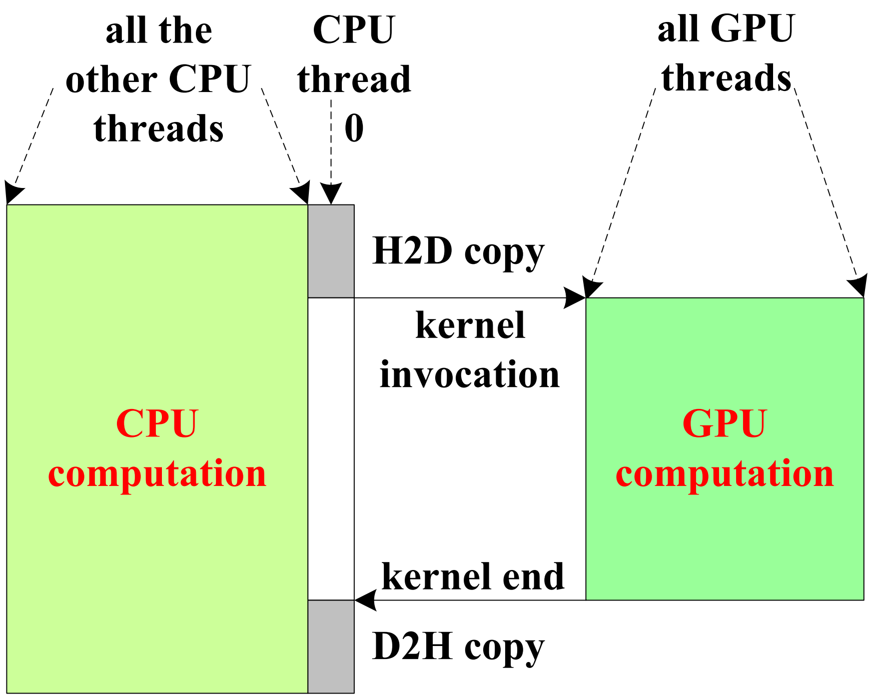


Fig. 2 The first CPU-GPU cooperative computing method

Main idea:

CPU thread 0 is used to control and to communicate with the GPU, all the other CPU threads together with all GPU threads to cooperatively perform some tasks.

Typical flow:

(1) CPU thread 0 firstly transfers part of the input data from CPU to GPU, next it invokes the CUDA kernel, then all GPU threads run the kernel in parallel, finally CPU thread 0 transfers the output data from GPU to CPU.

(2) At the same time, all the other CPU threads process the input data allocated to the CPU in parallel.

The CPU-GPU Cooperative Computing Environment

2. The second CPU-GPU cooperative computing method

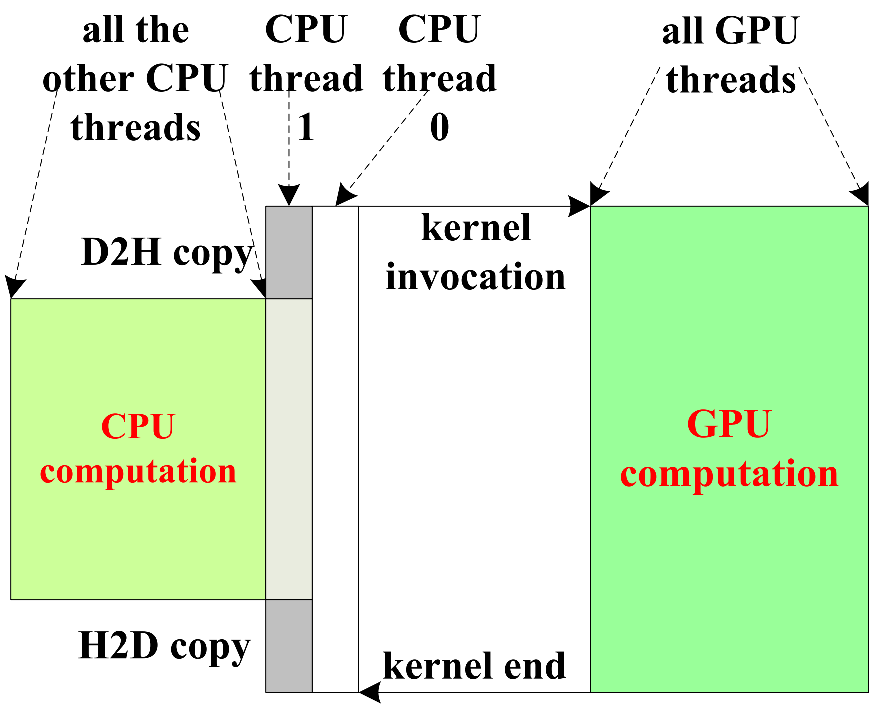


Fig. 3 The second CPU-GPU cooperative computing method

Main idea:

The main idea of the second method is similar to that of the first method.

Typical flow:

(1) CPU thread 0 is used to control the GPU, after it invokes the CUDA kernel, all GPU threads run the kernel in parallel.

(2) At the same time, CPU thread 1 firstly transfers part of the input data from GPU to CPU, then all the other CPU threads together with it to perform some tasks in parallel, finally CPU thread 1 transfers the output data from CPU to GPU.

The difference between the two methods:

Method 1: the data to be processed by the GPU comes from the CPU main memory

Method 2: the data to be processed by the CPU comes from the GPU global memory

The CPU-GPU Cooperative Computing Environment

3. The optimal task distribution model

- **The goal of establishing the model**

find the most appropriate task distribution ratio between CPU and GPU

- **The determination of the task distribution ratio needs to consider the following factors**
  - processing capabilities and memory capacities of the CPU side
  - processing capabilities and memory capacities of the GPU side
  - the bandwidth from CPU to GPU
  - the bandwidth from GPU to CPU
  - the actual time to run the given program only on the CPU
  - the actual time to run the given program only on the GPU
  - the CPU-GPU communication overhead

The CPU-GPU Cooperative Computing Environment

3. The optimal task distribution model

- **Some parameters used in the model**

Table 2. Parameters used in the task distribution model

Notation	Description
D	the total workload
D_{cpu}	the workload assigned to the CPU
D_{gpu}	the workload assigned to the GPU
R	the proportion of the workload assigned to the CPU
T_{cpu}	the running time of the CPU-only implementation
T_{gpu}	the running time of the GPU-only implementation
T_{comm}	the CPU-GPU communication time

The CPU-GPU Cooperative Computing Environment

3. The optimal task distribution model

- **The calculation of the task distribution ratio**

For the first method, the task distribution ratio can be calculated as follows:

$$R = \frac{T_{gpu} + T_{comm}}{T_{cpu} + T_{gpu} + T_{comm}}$$

For the second method, the task distribution ratio can be calculated as follows:

$$R = \frac{T_{gpu}}{T_{cpu} + T_{gpu} + T_{comm}}$$

The CPU-GPU Cooperative Computing Environment

3. The optimal task distribution model

- **The determination of the cooperative computing method**

$$T_{cpu} \begin{cases} <= T_{gpu}, \text{ adopt the first cooperative computing method} \\ > T_{gpu}, \text{ adopt the second cooperative computing method} \end{cases}$$

For the parallel two-list algorithm, the experimental results show that $T_{cpu} > T_{gpu}$, so **we adopt the second method**.

The CPU-GPU Cooperative Implementation

1. Three stages of the parallel two-list algorithm

The parallel generation stage

Firstly get ***n***-element input vector ***W*** and divide it into two equal parts W_1 and W_2, then generate two sorted lists ***A*** and ***B***.

The parallel pruning stage

At first evenly divide lists ***A*** and ***B*** into ***k*** blocks, then k^2 block pairs are assigned to ***k*** threads, finally use the ***prune rule*** to shri nk the search space of each thread.

The parallel search stage

At first those picked block pairs are evenly assigned to ***k*** threads, then each thread perfo rms the search routine to find a solution of SSP.

The CPU-GPU Cooperative Implementation

2. The cooperative implementation of the generation stage

Algorithm 1 The cooperative implementation of the generation stage

Require: W_1, W_2, M

for $i = 2$ **to** $n/2$ **do**

  ▷ **the add item process**

  Determine the task distribution ratio of the add item process.

  Determine the workload of the CPU and GPU during the add item process.

  Execute the add item process on both the CPU and GPU sides.

  ▷ **the partition and merge processes**

  Determine the task distribution ratio of the partition and merge processes.

  Determine the workload of the CPU and GPU during the partition and merge processes.

  Execute the partition and merge processes on both the CPU and GPU sides.

end for

return the sorted list A (B)

The CPU-GPU Cooperative Implementation

2. The cooperative implementation of the generation stage

The whole process of generating the sorted list ***A*** (***B***) needs to execute ***n*/2-1** iterations to complete. Each iteration mainly consists of the following six steps:

Step 1: Determine the task distribution ratio of the add item process

Step 2: Determine the workload of the CPU and GPU during the add item process

Step 3: Execute the add item process on both the CPU and GPU sides

Step 4: Determine the task distribution ratio of the partition and merge processes

Step 5: Determine the workload of the CPU and GPU during the partition and merge processes

Step 6: Execute the partition and merge processes on both the CPU and GPU sides

The CPU-GPU Cooperative Implementation

3. The cooperative implementation of the pruning and search stages

Algorithm 2 The cooperative implementation of the pruning and search stages

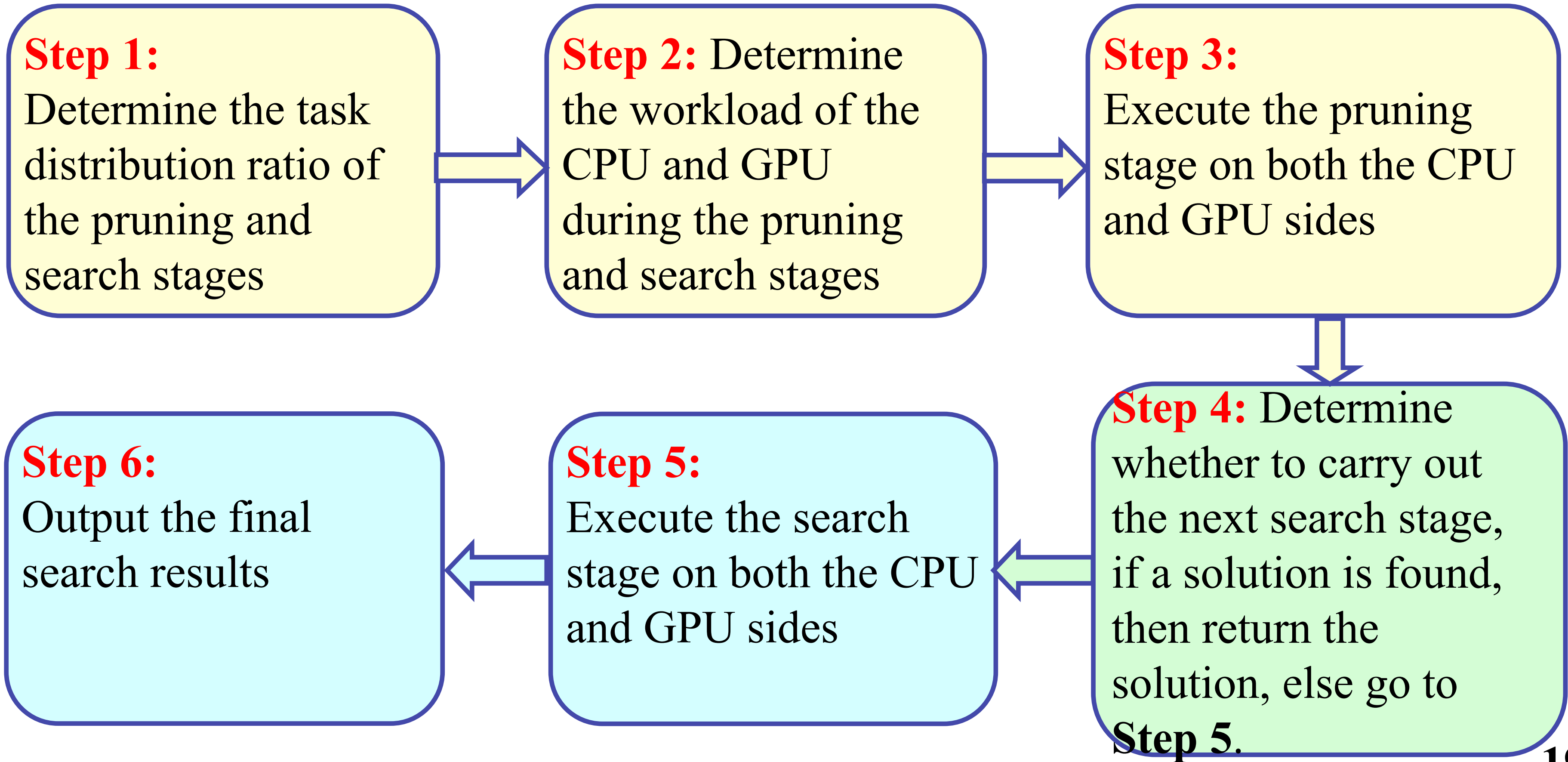
Require: ***A***, ***B***, ***M***

Determine the task distribution ratio of the pruning and search stages.
Determine the workload of the CPU and GPU during the pruning and search stages.
Execute the pruning stage on both the CPU and GPU sides.
if a solution of SSP is found **then**
  return the solution of SSP;
else
  Execute the search stage on both the CPU and GPU sides.
end if
if a solution of SSP is found **then**
  return the solution of SSP;
else
  return NULL; ▷ there is no solution
end if

The CPU-GPU Cooperative Implementation

3. The cooperative implementation of the pruning and search stages

The CPU-GPU cooperative implementation of the pruning and search stages mainly consists of the following six steps:



Experimental Evaluation

1. Experimental setup

- **Two different test platforms**

Table 3. Two different test platforms

	Test Platform 1	**Test Platform 2**
CPU	**Dual 4-cores** Intel Xeon E5504 CPUs (2.0 GHz)	**Dual 6-cores** Intel Xeon E5-2620 CPUs (2.0 GHz)
GPU	An NVIDIA **GTX 465** GPU (**352** CUDA **cores** at 607 MHz)	An NVIDIA **Tesla M2090** GPU (**512** CUDA **cores** at 1.3 GHz)
CPU main memory	**32 GB**	**32 GB**
GPU global memory	**1 GB,** 102.6 GB/s memory bandwidth	**6 GB,** 177.6 GB/s memory bandwidth
software	SUSE Linux Enterprise 11 operating system with NVIDIA **CUDA** driver version **5.5** and **GCC** version **4.4.7**	

Experimental Evaluation

1. Experimental setup

■ Three different parallel implementations

We use three different methods to implement the parallel two-list algorithm

CPU-only implementation	GPU-only implementation	CPU-GPU cooperative implementation
Implement the parallel algorithm **on CPU** using **OpenMP**	Implement the parallel algorithm **on GPU** using **CUDA**	Implement the parallel algorithm **on both CPU and GPU** using **OpenMP** and **CUDA**

■ Seven different problem sizes

1) **the problem size *n* = 42, 44, 46, 48, 50, 52, 54**
2) for each problem size, we randomly produce 100 different instances of SSP.
3) the average execution time of 100 instances is considered, and it is measured in **milliseconds.**

Experimental Evaluation

2. Evaluation of the task distribution model

- **The estimated task distribution ratios**

Table 4. The estimated task distribution ratio of the partition and merge processes

n	Test Platform 1				Test Platform 2			
	Tcpu	*Tgpu*	*Tcomm*	R_1	*Tcpu*	*Tgpu*	*Tcomm*	R_1
42	92.4	67.1	88.2	**27.09%**	70.2	56.6	73.5	**28.25%**
44	150.8	102.5	138.7	**26.15%**	114.3	85.4	115.6	**27.10%**
46	255.8	168.0	231.6	**25.63%**	194.1	140.1	193.0	**26.57%**
48	454.8	293.1	419.8	**25.10%**	344.9	244.2	349.9	**26.01%**
50	851.3	540.0	797.4	**24.67%**	645.3	449.4	664.5	**25.54%**
52	1626.3	1025.6	1509.1	**24.65%**	1236.8	854.8	1257.5	**25.52%**
54	3273.1	2059.2	3027.5	**24.63%**	2492.4	1717.0	2522.9	**25.50%**

Table 5. The estimated task distribution ratio of the pruning and search stages

n	Test Platform 1				Test Platform 2			
	Tcpu	*Tgpu*	*Tcomm*	R_2	*Tcpu*	*Tgpu*	*Tcomm*	R_2
42	18.0	6.3	9.4	**18.78%**	13.5	5.3	8.4	**19.34%**
44	29.1	10.0	18.3	**17.43%**	21.7	8.4	15.1	**18.52%**
46	48.7	15.8	33.3	**16.19%**	36.5	13.2	24.8	**17.73%**
48	85.3	27.2	58.7	**15.86%**	63.9	22.6	49.5	**16.63%**
50	156.8	49.5	110.1	**15.65%**	117.3	41.2	98.6	**16.02%**
52	293.5	92.2	207.3	**15.55%**	220.3	76.9	185.4	**15.94%**
54	576.0	180.5	410.0	**15.48%**	432.8	150.5	369.7	**15.79%**

Experimental Evaluation

2. Evaluation of the task distribution model

- **Verify whether the estimated task distribution ratio is reasonable**

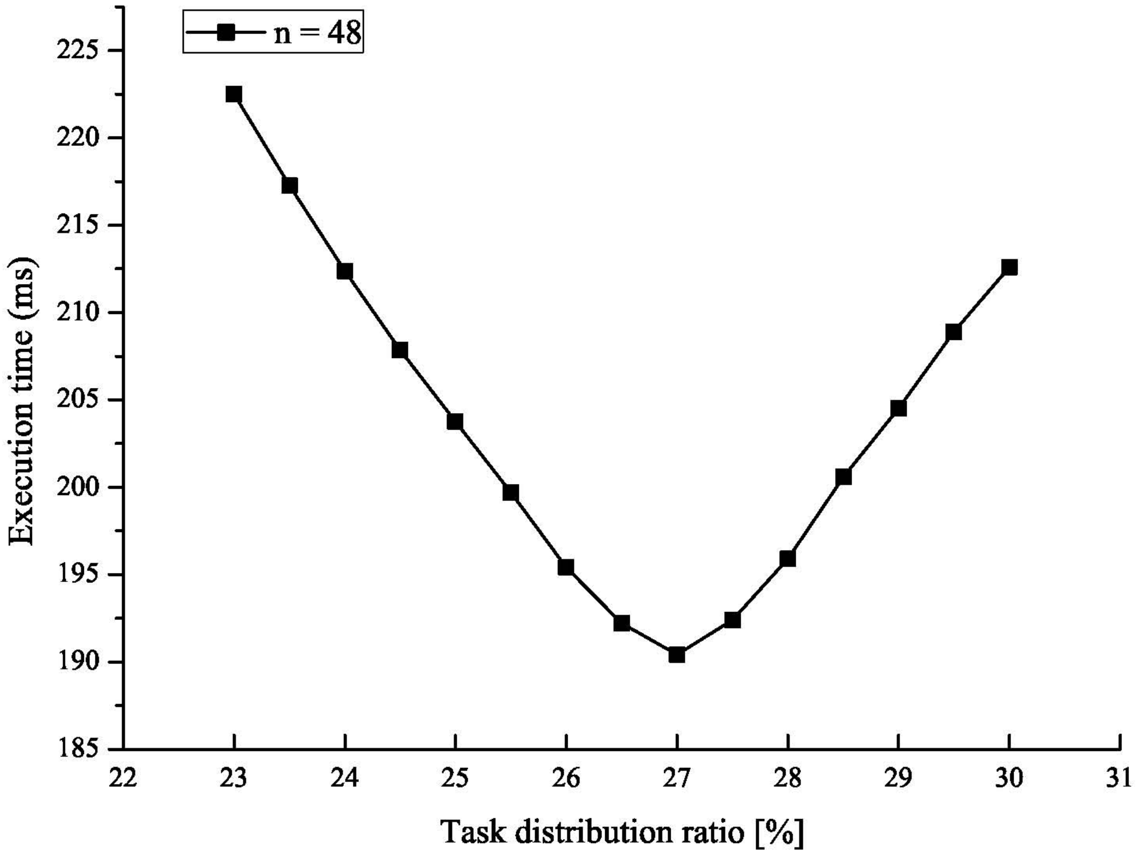


Fig. 4 The execution time of the partition and merge processes for different task distribution ratios on Test Platform 2 for ***n*** = 48

We specify the problem size ***n* = 48** and test the execution time of the partition and merge processes by using different distribution ratios on Test Platform 2.

Figure 4 shows that the estimated task distribution ratio has **only 1% error.** Hence, the error is acceptable.

Experimental Evaluation

2. Evaluation of the task distribution ratio

- **The actual optimal task distribution ratios**

Table 6. The actual optimal task distribution ratio for different problem sizes on two different test platforms

n	Test Platform 1		Test Platform 2	
	R_1	R_2	R_1	R_2
42	28.12%	19.81%	29.28%	20.40%
44	27.17%	18.39%	28.13%	19.54%
46	26.64%	17.08%	27.59%	18.71%
48	26.10%	16.73%	27.02%	17.54%
50	25.66%	16.51%	26.54%	16.90%
52	25.64%	16.41%	26.52%	16.82%
54	25.62%	16.33%	26.49%	16.66%

The results show that the estimated task distribution ratios are close to the actual optimal values, so **our proposed task distribution model can find reasonable task distribution ratio.**

Experimental Evaluation

3. Performance evaluation of the CPU-GPU cooperative implementation

Table 7. The execution times and speedups of three different parallel implementations on Test Platform 1

n	Sequential	Parallel implementation					
		CPU-only		GPU-only		CPU + GPU	
	Time	Time	Speedup	Time	Speedup	Time	Speedup
42	341.3	114.3	2.99	77.6	4.40	61.5	5.55
44	613.8	186.2	3.30	118.5	5.18	94.2	6.52
46	1094.2	315.4	3.47	193.7	5.65	154.4	7.09
48	2020.5	559.5	3.61	337.3	5.99	270.9	7.46
50	3868.3	1044.5	3.70	620.9	6.23	501.8	7.71
52	7513.4	1989.8	3.78	1177.7	6.38	951.9	7.89
54	15197.1	3990.8	3.81	2359.8	6.44	1907.6	7.97

Here, the *speedup* is defined as *sequential execution time* over *parallel execution time*.

Experimental Evaluation

3. Performance evaluation of the CPU-GPU cooperative implementation

Table 8. The execution times and speedups of three different parallel implementations on Test Platform 2

n	Sequential	Parallel implementation					
		CPU-only		GPU-only		CPU + GPU	
	Time	Time	Speedup	Time	Speedup	Time	Speedup
42	327.1	89.5	3.66	67.4	4.85	52.9	6.18
44	588.1	145.4	4.04	99.2	5.93	78.4	7.50
46	1047.7	246.6	4.25	161.4	6.49	128.2	8.17
48	1933.7	437.3	4.42	281.1	6.88	224.1	8.63
50	3700.1	816.0	4.53	516.8	7.16	413.4	8.95
52	7182.8	1559.4	4.61	982.6	7.31	786.7	9.13
54	14520.8	3131.7	4.64	1967.6	7.38	1577.3	9.21

Experimental Evaluation

3. Performance evaluation of the CPU-GPU cooperative implementation

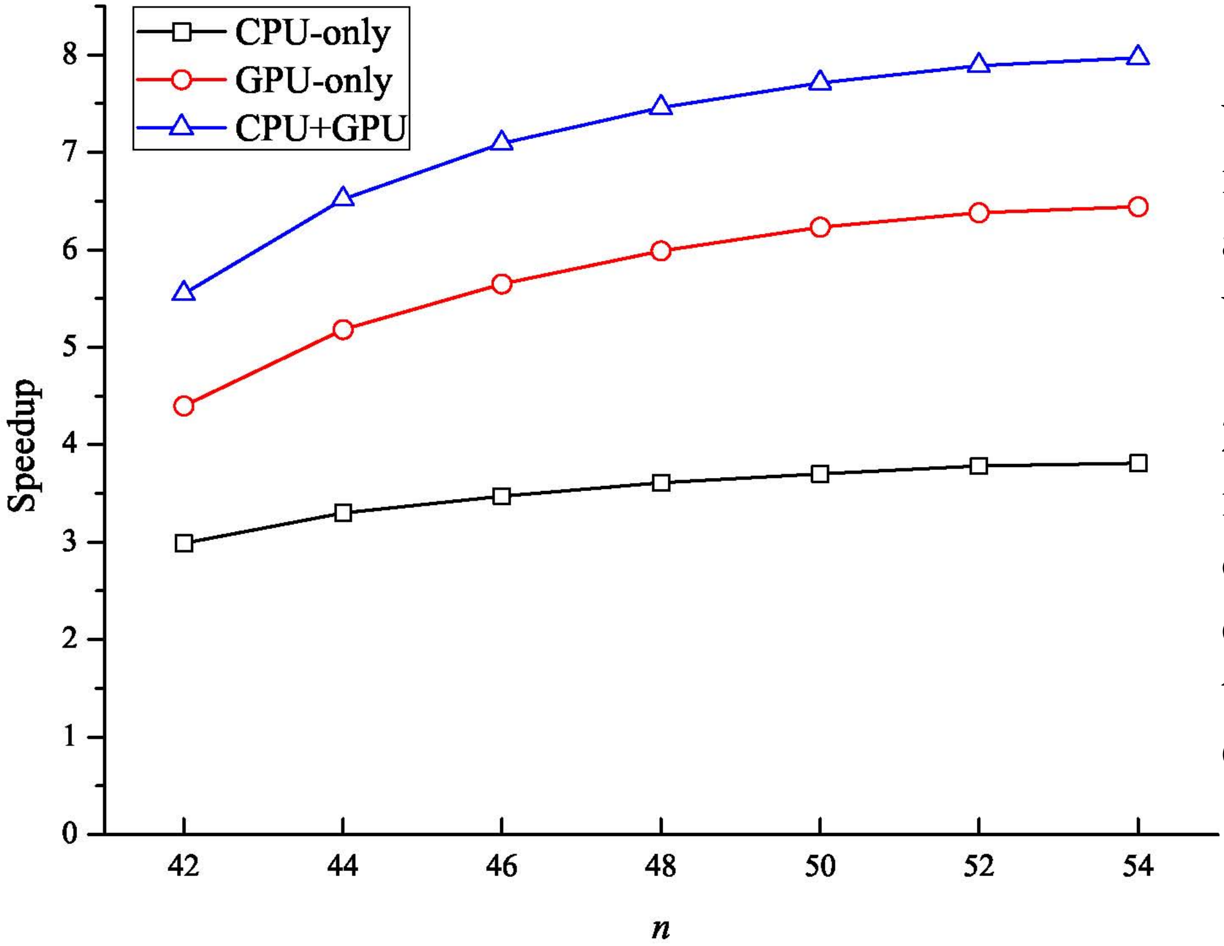


Fig. 5 The **speedup comparison** of three different parallel implementations **on Test Platform 1**

1) The results show that when the problem size increases, the speedups grow accordingly, and the speedup will gradually reach a peak.

2) The CPU-GPU cooperative implementation significantly outperforms the CPU-only case and the GPU-only case, this is because both CPU and GPU have been fully utilized.

Experimental Evaluation

3. Performance evaluation of the CPU-GPU cooperative implementation

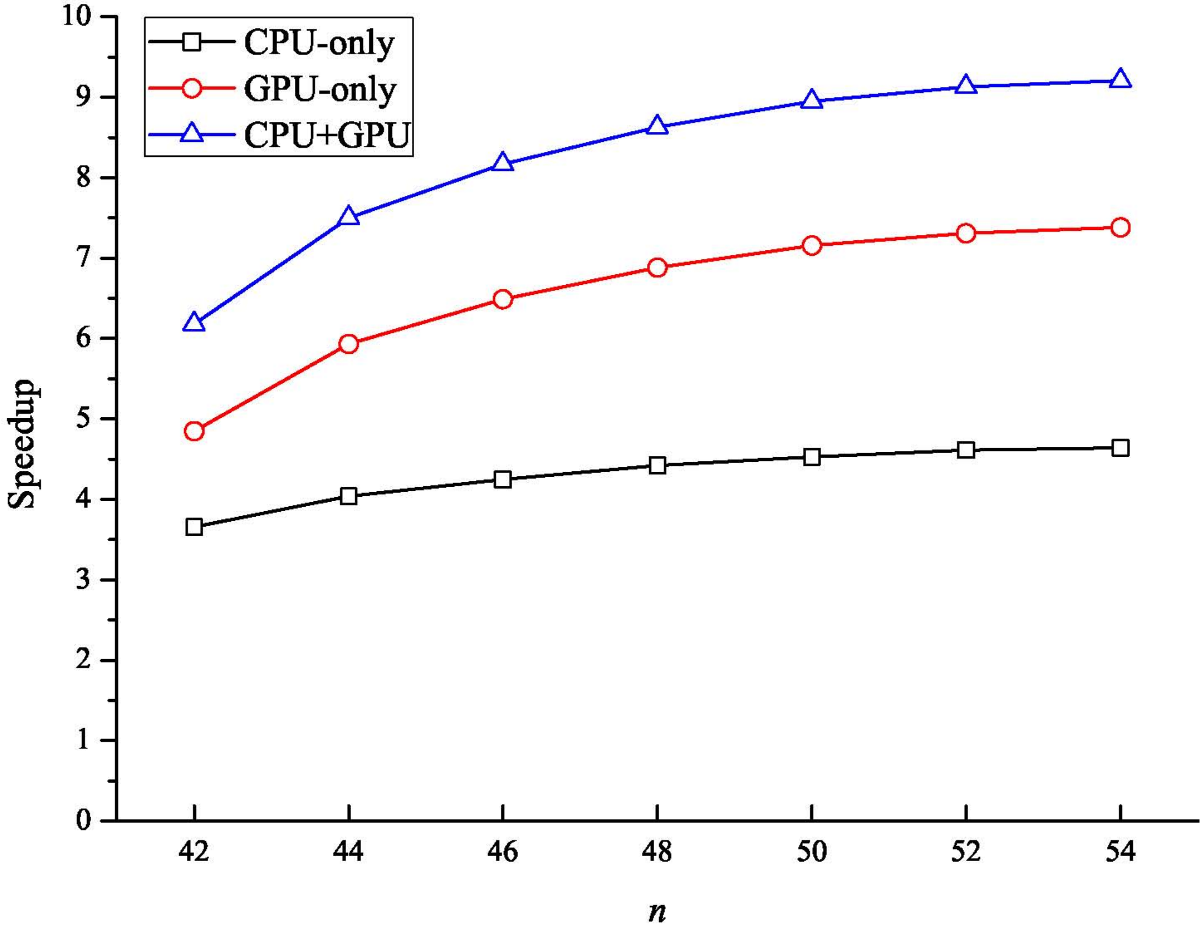


Fig. 6 The **speedup comparison** of three different parallel implementations **on Test Platform 2**

1) Figures 5 and 6 show that the CPU-GPU cooperative implementation **achieves substantial speedup,** when ***n* = 54**, it obtains **8 times speedup on Test Platform 1** and **9.2 times speedup on Test Platform 2.**

2) The results also show that Test Platform 2 produces better performance than Test Platform 1, indicating that **our approach has good scalability**.

Experimental Evaluation

3. Performance evaluation of the CPU-GPU cooperative implementation

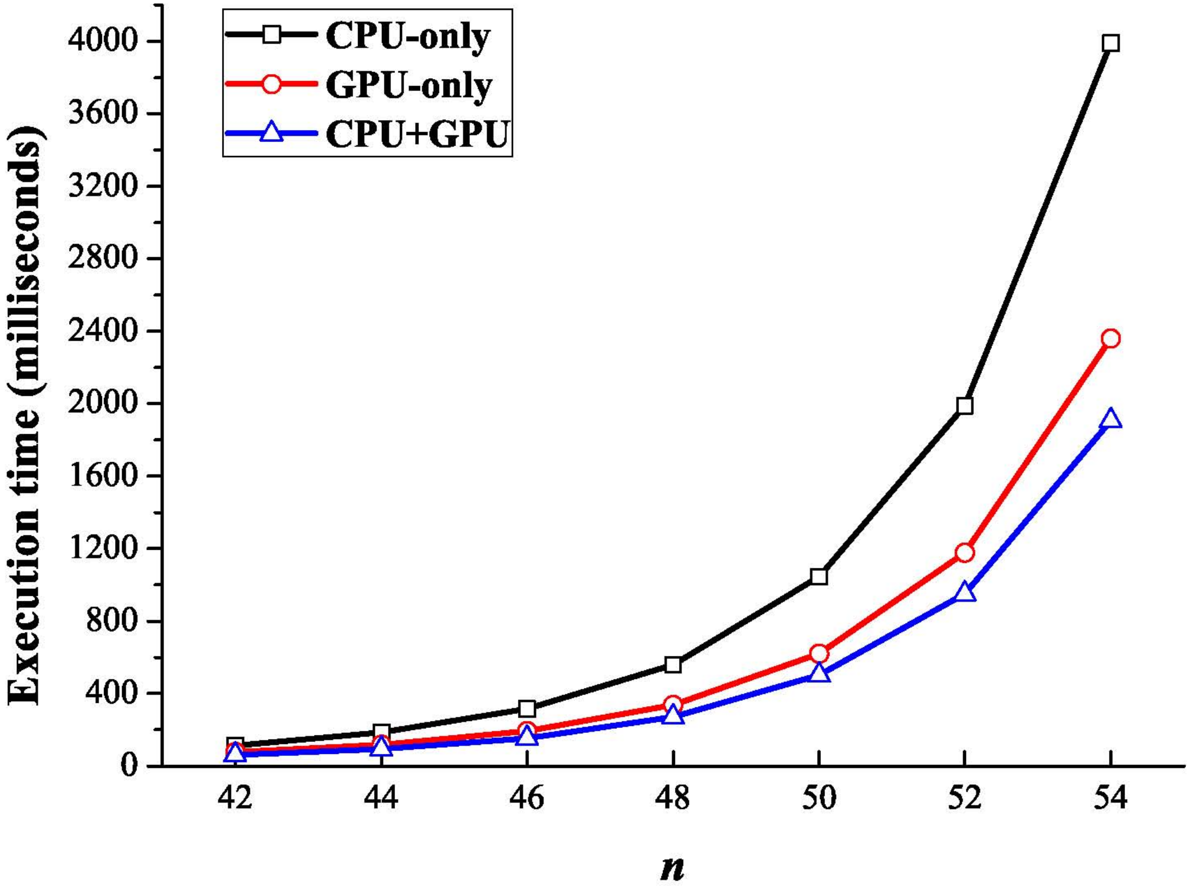


Fig. 7 The **execution time comparison** of three different parallel implementations **on Test Platform 1**

1) **Compared with the CPU-only implementation,** the CPU-GPU cooperative implementation achieves **an average of 103% performance improvement** on Test Platform 1.

2) **Compared with the GPU-only implementation,** the CPU-GPU cooperative implementation achieves **an average of 25% performance improvement** on Test Platform 1.

Experimental Evaluation

3. Performance evaluation of the CPU-GPU cooperative implementation

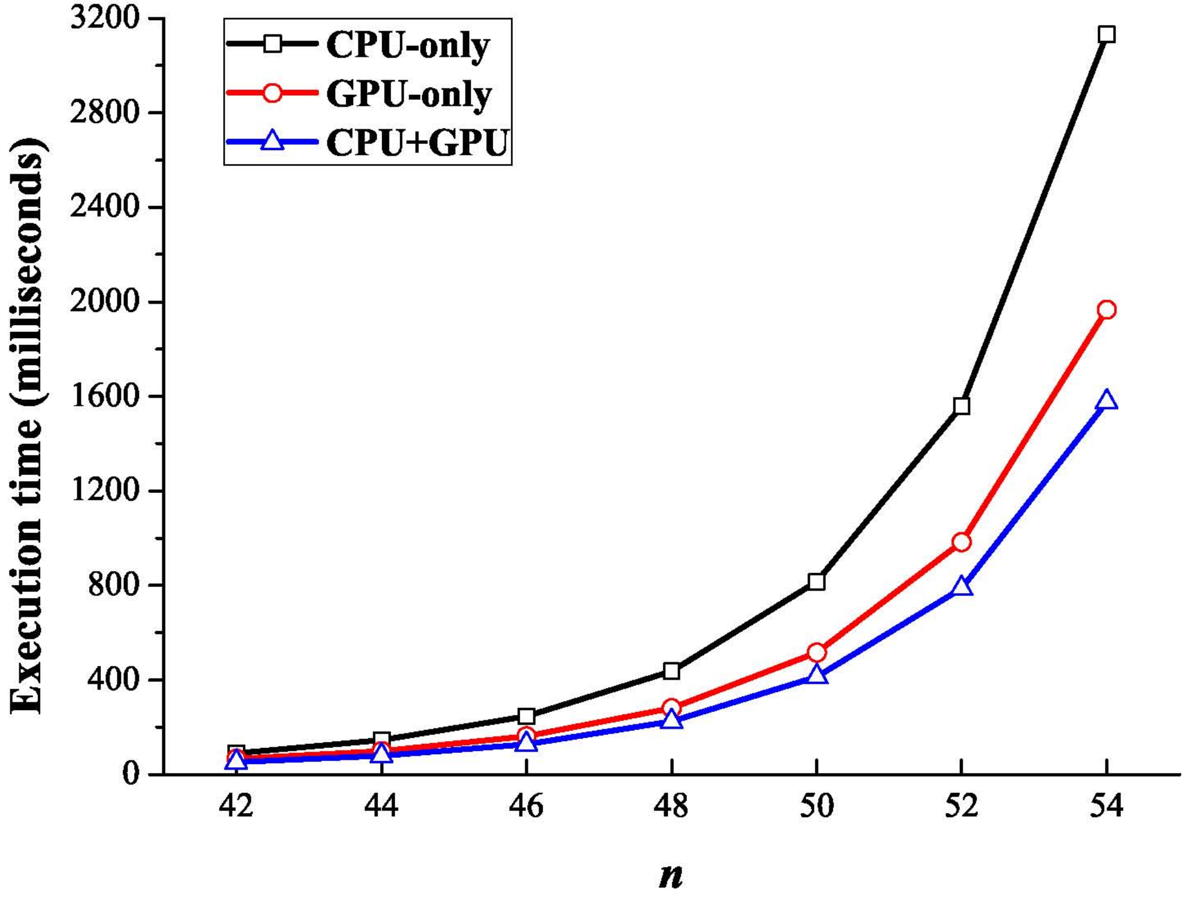


Fig. 8 The **execution time comparison** of three different parallel implementations **on Test Platform 2**

1) **Compared with the CPU-only implementation,** the CPU-GPU cooperative implementation achieves **an average of 91% performance improvement** on Test Platform 2.

2) **Compared with the GPU-only implementation,** the CPU-GPU cooperative implementation achieves **an average of 26% performance improvement** on Test Platform 2.

Conclusions

■ The main contributions

- ➢ An original CPU-GPU cooperative implementation of the parallel two-list algorithm is proposed to effectively solve the subset-sum problem.
- ➢ An optimal task distribution model is established to find the most appropriate task distribution ratio between CPU and GPU.

■ The future work

We will focus on the two performance bottlenecks:

- ➢ the communication overhead between CPU and GPU
- ➢ the load balance between CPU and GPU

Thanks!