Hindawi
Complexity
Volume 2021, Article ID 6694944, 15 pages
https://doi.org/10.1155/2021/6694944

Research Article

Parallel Implementations of Candidate Solution Evaluation Algorithm for N-Queens Problem

Jianli Cao, [1] **Zhikui Chen**, [1] **Yuxin Wang**, [2] **and He Guo** [1]

[1] *School of Software Technology, Dalian University of Technology, Dalian, China*
[2] *School of Computer Science and Technology, Dalian University of Technology, Dalian, China*

Correspondence should be addressed to Jianli Cao; cjllean@mail.dlut.edu.cn

Received 29 November 2020; Revised 4 January 2021; Accepted 28 January 2021; Published 22 February 2021

Academic Editor: Leo Y. Zhang

Copyright © 2021 Jianli Cao et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

The N-Queens problem plays an important role in academic research and practical application. Heuristic algorithm is often used to solve variant 2 of the N-Queens problem. In the process of solving, evaluation of the candidate solution, namely, fitness function, often occupies the vast majority of running time and becomes the key to improve speed. In this paper, three parallel schemes based on CPU and four parallel schemes based on GPU are proposed, and a serial scheme is implemented at the baseline. The experimental results show that, for a large-scale N-Queens problem, the coarse-grained GPU scheme achieved a maximum 307-fold speedup over a single-threaded CPU counterpart in evaluating a candidate solution. When the coarse-grained GPU scheme is applied to simulated annealing in solving N-Queens problem variant 2 with a problem size no more than 3000, the speedup is up to 9.3.

1. Introduction

The Eight-Queens problem was first proposed by Max Bezzel in a Berlin chess magazine in 1848 [1]. The original question was how to place eight queens on the chessboard and make them unable to attack each other. If the number of queens of the problem is expanded, it becomes the famous N-Queens problem. The N-Queens problem has many applications in real-world and theoretical research, such as artificial intelligence, graph theory, circuit design, air traffic control, data compression, and computer task scheduling [2]. The input of the N-Queens problem is only the number of queens. According to different requirements, the output can be the number of solutions or the sequence of each solution. Since the problem has been proved to be NP-hard, the only way to obtain the number of solutions is to find these solutions. Therefore, the amount of calculation required to obtain these two types of outputs is the same, and the only difference is whether each solution is saved in the calculation process. There are three variants of N-Queens problem according to the different demands for the number of valid solutions:

Variant 1: finding one valid solution for each problem size

Variant 2: finding a set of different valid solutions for larger problem scale

Variant 3: finding all valid solutions if the problem scale is small enough

Variant 1 has been solved in 1969. Hoffman [3] proposed a construction method by analyzing the inherent mathematical laws of the N-Queens problem. This method can obtain a valid constructive solution within the time complexity of $O(1)$. However, his construction method can only construct one fixed valid solution for each N value that is not universal.

Variant 3 was mathematically proven to be NP-hard, and there is no deterministic algorithm in polynomial time to solve all valid solutions. For variant 3 with small N value, brute force, backtracking, and tree-based search algorithm can be used to get all valid solutions. Somes [4] solve the problem for $N = 23$ by recursive backtracking algorithm; Kise et al. [5] solve the problem for $N = 24$ using MPI (Message-Passing Interface) on general-purpose processors; Caromel et al. [6] solve the problem for $N = 25$ by using a heterogeneous grid of 260 computers; and Preußer et al. [7] solve a 26-queens problem in 270 days using a cluster of 26 FPGAs (field programmable gate arrays); to the best of our knowledge, this is a current world record.

Based on the performance of current computers, it is not realistic to find and save all valid solutions of the large-scale ($N > 26$) N-Queens problem in terms of running time and storage space. Therefore, for N-Queens problem variant 3, some researchers study the case of N greater than 26 by building devices with higher computing power, while others design new parallel algorithms to accelerate the case of $N < 26$.

Before the dynamic parallel technology appeared on GPU, the usual way to solve the combinatorial optimization problem with the backtracking algorithm implemented by GPU was to divide the problem into two steps: first, the precalculated subsearch trees are generated on the CPU, and then, these subtrees are assigned to the GPU thread to complete the further search [8–10]. Amrasinghe et al. [11] use NVIDIA Cg language to solve the N-Queens problem on GeForce 6800 hardware, but the performance of their algorithm is proved to be lower than that of Pentium M CPU with 2.0 GHz frequency. Pamplona et al. [12] design an N-Queens problem solver running on GeForce 9600 by using CUDA 1.0. The performance of their algorithm is also lower than that of the C++ implementation on an Intel quad core processor with 2.4 GHz frequency. Feinbube et al. [10] transplant Somers' algorithm to GPU for parallelization and use four optimization methods such as using shared memory to improve the performance of his algorithm. Their parallel algorithm is used to speed up the solving process of an N-Queens problem with a size range from 14 to 17 on GPU devices with a computing capability of 1.1 and 1.3 (GTX275, GTX295, NVS295, and GeForce8600M). Zhang et al. [13] optimize a GPU-based N-Queens solver by increasing L1 cache configuration, reducing shared memory bank conflicts, balancing thread load, etc. and obtain more than 10 times speedup with the number of queens ranging from 15 to 19 on GTX480. Thouti et al. [14] use the OpenCL programming model to analyze and solve the issues of atomicity and synchronization and obtained speedup of 20X with the number of queens between 16 and 21 on the Quadro FX 3800. Plauth et al. [15] use CDP (CUDA dynamic parallel) technology to solve the N-Queens problem. In his scheme, the kernel in each layer of the CDP recursive stack is responsible for one row or multiple rows of the chessboard. Plauth uses his scheme to solve the N-Queens problem with the problem size between 8 and 16 on Tesla K20, and the experimental results show that the performance of his scheme is lower than that of Feinbube's GPU non-CDP-based scheme and even lower than that of Somers' serial scheme in some cases. Carneiro et al. [16, 17] use a CDP-based backtracking algorithm to solve N-Queens problem variant 3 with a size ranging from 10 to 17. He concluded that CDP is less dependent on parameter tuning, but due to the high cost imposed by dynamically launched kernels, the performance of the CDP-based scheme is outperformed by non-CDP bitset-based implementation with well-tuned parameters and multicore counterparts.

The methods used to solve variant 3 can also be used to solve variant 2 to obtain a set of different solutions, but the efficiency is very low because these algorithms need to ensure that every position in the solution space is searched without omission. In addition, because these algorithms search all the solution spaces in a certain order, the solutions are likely to be located in the close position in the solution space, and the diversity of solutions is not strong enough. So, variant 2 is usually solved by heuristic algorithm and random algorithm.

The output of variant 2, a set of valid solutions for the large-scale N-Queens problem, can be used in scientific research and practice. For example, we want the neural network to have the ability to generate zero conflict or less conflict N-Queens layout, and the output of variant 2 can provide a set of solutions to the neural network as a training sample set. In circuit design, for the reason of signal interference, or in arts and crafts, just for the sake of beauty, it may be required that devices and patterns cannot be placed in the same line, column, or diagonal. In this case, the output of variant 2 can complete this requirement.

Variant 2 can be regarded as a permutation-based combinatorial optimization problem or a constraint satisfaction problem, and researchers often get valid solutions by using random algorithm and heuristic algorithm. The process of solving is to generate a group of random solutions first, then guide these candidate solutions to evolve in a better direction through various heuristic information, and finally, get the optimal solution. Hu Xiaohui et al. [18] use the improved PSO (Particle Swarm Optimization) algorithm to obtain part of the valid solutions with $N \leq 200$; Jafarzadeh et al. [19] use PSO and SA (Simulated Annealing) to obtain part of the valid solutions with $N \leq 2000$; Zhang et al. [20] use CRO (Chemical Reaction Optimization) algorithm to solve an eight-queens problem; Hu Nengfa et al. [21] use simplified GA (Genetic Algorithm) to calculate a valid solution for $N = 500$ in 45 seconds; Zhang Buzhong et al. [22] implement an operator-oriented parallel genetic algorithm in the multicore platform for solving $N = 1500$ in 20655 seconds; Turky et al. [23] use the genetic algorithm to obtain a valid solution with $N = 2000$ in 9123 seconds; Wang et al. [24] use four core i5 processors to implement the parallel genetic algorithm and achiev a speedup of 2.8 compared with a serial counterpart when the problem scale reached 512; and Cao et al. [25] constructed a two-level parallel genetic algorithm based on a GPU cluster, which expands the N-Queens solution scale to 10000 in the acceptable time.

Those heuristic algorithms need to evaluate candidate solutions generated in the search process. The number of queens with conflicts in the candidate solutions is an appropriate evaluation criterion. The conflict number can be calculated with the following formulas:

$$\text{conflicts} = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} g(i,j), \tag{1}$$

$$g(i,j) = \begin{cases} 1, & \begin{array}{l}\text{if } NQ_i == NQ_j \\ \text{or } \left|NQ_j - NQ_i\right| == j - -i,\end{array} \\ 0, & \text{otherwize.} \end{cases} \tag{2}$$

This calculation process is usually encapsulated as an evaluation function. In other works, it is also called cost function, objective function, or fitness function. This function has the time complexity of $O(N^2)$ and high parallelism. Because it needs to be executed repeatedly, this function takes up a lot of running time in the heuristic algorithm.

Simulated annealing [26] is a kind of heuristic algorithm which simulates the process that metals tend to be stable during heating and cooling in metallurgy. Simulated annealing algorithm has been proved to be asymptotically convergent and can converge to the global optimal solution with probability 1 under the condition that the initial temperature is high enough, the cooling speed is slow enough, and the termination temperature is low enough. Because the simulated annealing algorithm is simple and has few control parameters, we use this algorithm to demonstrate the acceleration effect of parallelization of evaluation function on the whole algorithm.

Because the time cost to ensure the algorithm converges to the valid solution with probability 1 is too high, we use the parameters in Table 1 to get the result with a probability higher than 0.5. With this set of parameters, the algorithm can get a valid solution in a few hours. The average running time of the algorithm is 9443 seconds with problem size 3000. We use std: : shuffle to shuffle the sequence from 1 to N to get the initial random solution.

Figure 1 depicts the running time proportion of the evaluation function in the whole simulated annealing algorithm for different N values.

It can be seen from Figure 1 that the larger the N, the higher the proportion of the evaluation function. When N is greater than 700, the proportion of evaluation function exceeds 99%. Therefore, for heuristic algorithms based on search and evaluation, such as simulated annealing, genetic algorithm, and chemical reaction optimization, accelerate evaluation function is the key to improving the speed of the whole algorithm in solving a large-scale N-Queens problem.

Since evaluation function has high parallelism and simple operation, it is very suitable for GPU (graphical processing unit), which uses the SIMT(Single Instruction Multiple Threads) model, originally designed for graphics applications and optimized for high throughput by allocating more transistors to compute unit, instead cache, prediction unit, etc.

Our objective is to speed up the search and evaluation-based heuristic algorithm in solving variant 2 of the N-Queens problem by accelerating the fitness function. The main acceleration method is the thread-level parallel technology of CPU and GPU. In this paper, we propose four GPU-based parallel schemes by using CUDA 8.0 [27] parallel technology with different parallel granularities and three CPU-based parallel schemes by using C++ multithreading technology, Intel TBB library, and Java Fork-Join framework. Performances of these schemes are verified through experiments. The scheme with the highest speedup is applied to simulated annealing algorithm for accelerating N-Queens problem variant 2.

The organization of this paper is as follows. In Section 1, we introduce three variants of the N-Queens problem and related research. The significance of improving the performance of the evaluation function is also discussed here through an experiment. In Section 2, one CPU-based serial scheme, three CPU-based parallel schemes, and four GPU-based parallel schemes of realizing evaluation functions are proposed. In Section 3, we compare the performance of the first seven schemes, and the eighth scheme is also discussed separately here. In Section 4, the validity of the GPU-based coarse-grained scheme is verified by the simulated annealing algorithm. Section 5 discusses future work and concludes this paper.

2. Parallel Schemes of Fitness Function

An N-Queens problem is a two-dimensional optimization problem. In order to facilitate the mutation, crossover, synthesis, splitting, and other operations in the evolution process of the heuristic algorithm, the candidate solution is usually encoded by an integer and expressed as one-dimensional arrays or a vector. The subscripts of the array or vector are used as abscissas, and the element values are used as ordinates. For example, we use array $NQ = \{2,4,1,3\}$ to represent a candidate solution $\{(1,2),(2,4),(3,1),(4,3)\}$ and variable N to store array length.

The initial solution is obtained by shuffling the number sequence from 1 to N with std: : shuffle function. After the heuristic algorithm improves these initial solutions according to the heuristic information, the candidate solutions are sent to the fitness function. The number of conflicts calculated by fitness function is returned to heuristic algorithm as evaluation result. This process is often repeated many times.

2.1. CPU Single-Threaded Scheme.

2.1. CPU Single-Threaded Scheme. With a single-threaded processor, the scheme has to compare all pairs of queens sequentially. This scheme is described in Algorithm 1 and only used as a baseline to calculate the speedup of other parallel schemes.

2.2. CPU-Adaptive Multithreaded Scheme. We use the class std: : thread introduced in C++11 to implement a CPU multithreaded scheme, the task of thread function is described in Algorithm 2. In order to avoid the high cost of atomic operation, we design a counter array with the same length as the number of threads to store the number of conflicts calculated by the corresponding thread. After all threads are finished, STL function accumulate is used to get the total conflicts number of all threads. Algorithm 3 describes the process of accumulating all conflicts. Limited by the number of cores, the most

Table 1: Parameter setting of simulated annealing algorithm.

N	Initial temperature	Termination temperature	Cooling coefficient	Search times in current temperature
4–99	3000	10–6	0.97	50
100–900	3000	10–6	0.98	100
1000–3000	3000	10–6	0.99	200

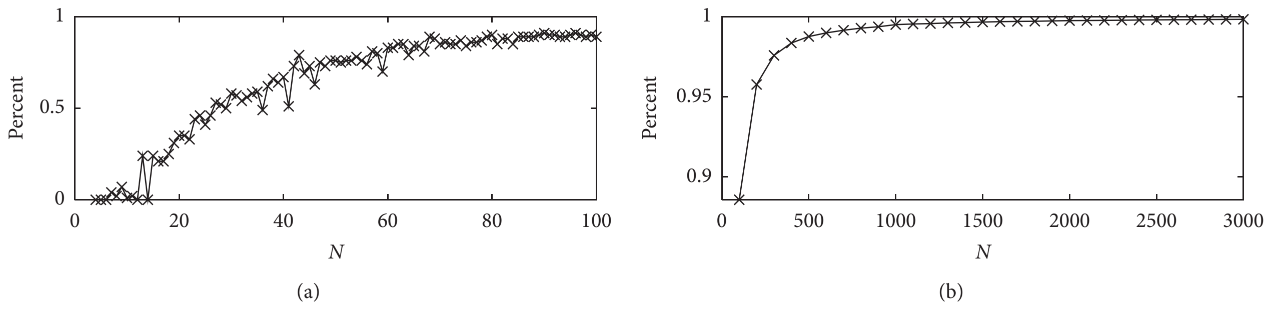

Figure 1: Time proportion of evaluation function in SA algorithm.

```
    Input: $N, NQ$
    Output: conflicts
(1) conflict ⟵ 0;
(2) for $i = 0$; $i$ ¡ $N$; ++I do
(3)     for $j = i + 1$; $j <$ N; ++$j$ do
(4)         if $NQ_i == NQ_j$ or $|NQ_j - NQ_i| ==$ j–i
    then
(5)             conflict++;
(6)         end
(7)     end
(8) end
```

Algorithm 1: Calculation conflicts with single-threaded CPU.

```
     Input: $N, NQ$, thr_$i\ dx$
     Output: conflict_arr
 (1) conflict_arr$_{\text{thr\_}i\ dx}$ ⟵ 0; /∗ clear counter ∗/
 (2) batchs ⟵ (N + thr_num -1)/thr_num;/∗ get task batchs ∗/
 (3) for int batch = 0; batch < batchs; batch++ do
 (4)     $i$ = thr_num * batch + thr_idx; /∗ get position $i$ ∗/
 (5)     for $j = i + 1$; $j < N$; ++$j$ do
 (6)         if $NQ_i == NQ_j$ or $|NQ_j - NQ_i| == j–i$
     then
 (7)             conflict_arr$_{\text{thr\_}i\ dx}$ ++;
 (8)         end
 (9)     end
(10) end
```

Algorithm 2: thread_function.

appropriate number of CPU threads is often less than 100, and the summation of the array can be performed quickly.

In this scheme, the number of threads can be set manually. We observe the performance of this scheme in different problem sizes when the number of threads varies from 1 to 60 and find that the optimal number of threads is related to the size of the problem. Figure 2 shows the speedup of this scheme compared with the single-threaded scheme with different thread numbers, and different colors in the legend indicate different problem sizes.

For the case of $N \leq 1000$, the maximum speedup is less than 5, and the optimal number of threads is about 10. If the number of threads participating in the calculation exceeds the optimal number, the cost of redundant threads is greater than the benefit and the performance will decrease.

For the case of $1000 < N \leq 10000$, the maximum speedup is between 10 and 15, and the optimal number of threads is about 20. For the cases of $N > 20000$, the maximum speedup is between 10 and 20, the optimal number of threads is about 40, which is the number of logical cores of our test platform. If the number of threads exceeds the optimal value, the performance will be only slightly affected because at this time, all processors are fully utilized, and increasing the number of threads will not continue to improve the utilization of the processors.

The maximum speedup and the corresponding optimal number of threads for each different problem scale are plotted in Figure 3. We observed that as the size of the problem increased, the speedup and the corresponding number of optimal threads gradually approached the number of cores. We store the optimal number of threads corresponding to each problem size in std: : map data structure. In the later experiments, this map is used to select the optimal number of threads for different scale problems, so that the algorithm has a certain adaptive ability.

2.3. CPU Intel TBB Scheme. Intel Threading Building Blocks (TBB) is a library that takes full advantage of multicore performance. The key notion of TBB is to separate logical task patterns from physical threads and to delegate task scheduling to the system. Compared with using the raw thread library, such as POSIX threads, std: : thread, or Boost threads, users can focus on the decomposition of tasks instead of allocating computation and data to threads manually and the synchronization issue among threads.

We use the function tbb: : parallel_reduce provided by TBB to decompose and summarize the calculation tasks of evaluation function. To make use of this function, we need to design a class and override function operator and join in the NQClass which is defined in Algorithm 4. The function SubTask() is used to complete the decomposed subtask, that is, calculate the number of conflicts caused by the queen i. This function is called infunction operator.

Block_range used in Algorithm 5 is a class defined by TBB in the file blocked_range.h to indicate the range of data to be processed. After the required class is defined, the use of function tbb: : parallel_reduce is very simple. Without specifying the number of threads manually, TBB can automatically decompose subtasks and complete them in parallel.

2.4. CPU Fork-Join Scheme. Fork/Join is a framework provided by JAVA7 for parallel task execution. By using job stealing technology to schedule tasks, the Fork/Join framework can achieve better load balancing among multiple cores. The key to implementing the evaluation function with this framework is to inherit class RecursiveTask and override function compute.

As shown in Algorithm 6, tasks larger than the threshold are divided into smaller tasks recursively and delivered to multiple cores for execution. We tried different thresholds and found that best performance can be achieved when the threshold is between 2 and 10. The experimental data used in the following section are obtained with threshold = 2, which means that each thread only calculates the number of conflicts caused by a single queen.

2.5. GPU Fine-Grained Scheme 1. Considering the SIMT structure of the GPU, one can run thousands of threads at the same time. To make full use of the number of threads in a fine-grained scheme, we use one thread to compare a pair of queens to ascertain if there is a conflict between them. We use the CPU to calculate the subscript pairs of queens that need to be compared, as shown in Algorithm 7. There are a total of $(N(N-1)/2)$ pairs of subscripts to be stored in array. The array is then transported along with the candidate solution to the GPU through the PCI-E data bus. In the GPU kernel, each GPU thread reads a pair of subscripts and extracts the location of the corresponding queen according to the subscript. If there is a conflict between the two, the atomic operation is used to add one to the counter in the global memory of GPU.

Array NQ and subscript array Pairs have been transferred from the host memory to the GPU memory by cudaMemcpy before kernel run. The task of each thread of GPU is described in Algorithm 8 which has $O(1)$ time complexity. Variable conflicts is a global variable that can be accessed by all threads.

When problem size N increases to 50000, approximately 1.164G pairs of queens need to be compared. If the subscript is saved with the unsigned short type, more than 4 GB memory is needed. The huge amount of data transfer between CPU and GPU takes up most of the running time, which completely offsets the benefits of parallel computing. The performance of this scheme is 2 to 4 orders of magnitude lower than that of the coarse-grained GPU scheme. As the scale of the problem increases, the disadvantage will continue to be magnified. Even if GPU can reduce the cost of data transmission by multistream and overlap of computation and data transmission, this scheme has few performance advantages. So, we did not continue to test the performance of this scheme for $N > = 50000$.

2.6. GPU Fine-Grained Scheme 2. Considering fine-grained scheme 1, the subscripts array is generated by the CPU and transferred to the GPU through the PCI-E bus. With the increase in problem size N, the memory consumption of subscript array increases at the speed of $O(N^2)$. To avoid the communication overhead caused by a large amount of data transmission between the CPU and GPU, in this scheme, the subscripts of the two queens that each GPU thread needs to compare is calculated by GPU thread according to its own index, The process of using GPU to generate subscript is shown in Algorithm 9. This scheme can improve the parallelism and improve the utilization of GPU resources. The part of the algorithm for calculating subscripts has $O(N)$ time complexity.

The disadvantage of this scheme is that the tasks of each thread are very few to give full play to make full use of GPU

```
    **Input:** $N$, $NQ[\,]$, thr_num$^{\dagger}$, conflict_arr$[\,]^{\dagger\dagger}$
    **Output**: conflicts
(1) for int $i = 0$; $i <$ thr_num; $i$++;/* start thr_num threads */
(2) do
(3)     thr_arr$_i \longleftarrow$ thread (thread_func,NQ,$N$,$i$,thr_num,conflict_arr)
(4) end
(5) for int $i = 0$; $i <$ thr_num; $i$++; /* wait for threads finish */
(6) do
(7)     thr_arr$_i$.join()
(8) end
(9) conflicts $\longleftarrow$ accumulate (conflict_arr, conflict_arr + thr_num, 0)
```

ALGORITHM 3: Calculation of conflicts with multithreaded.

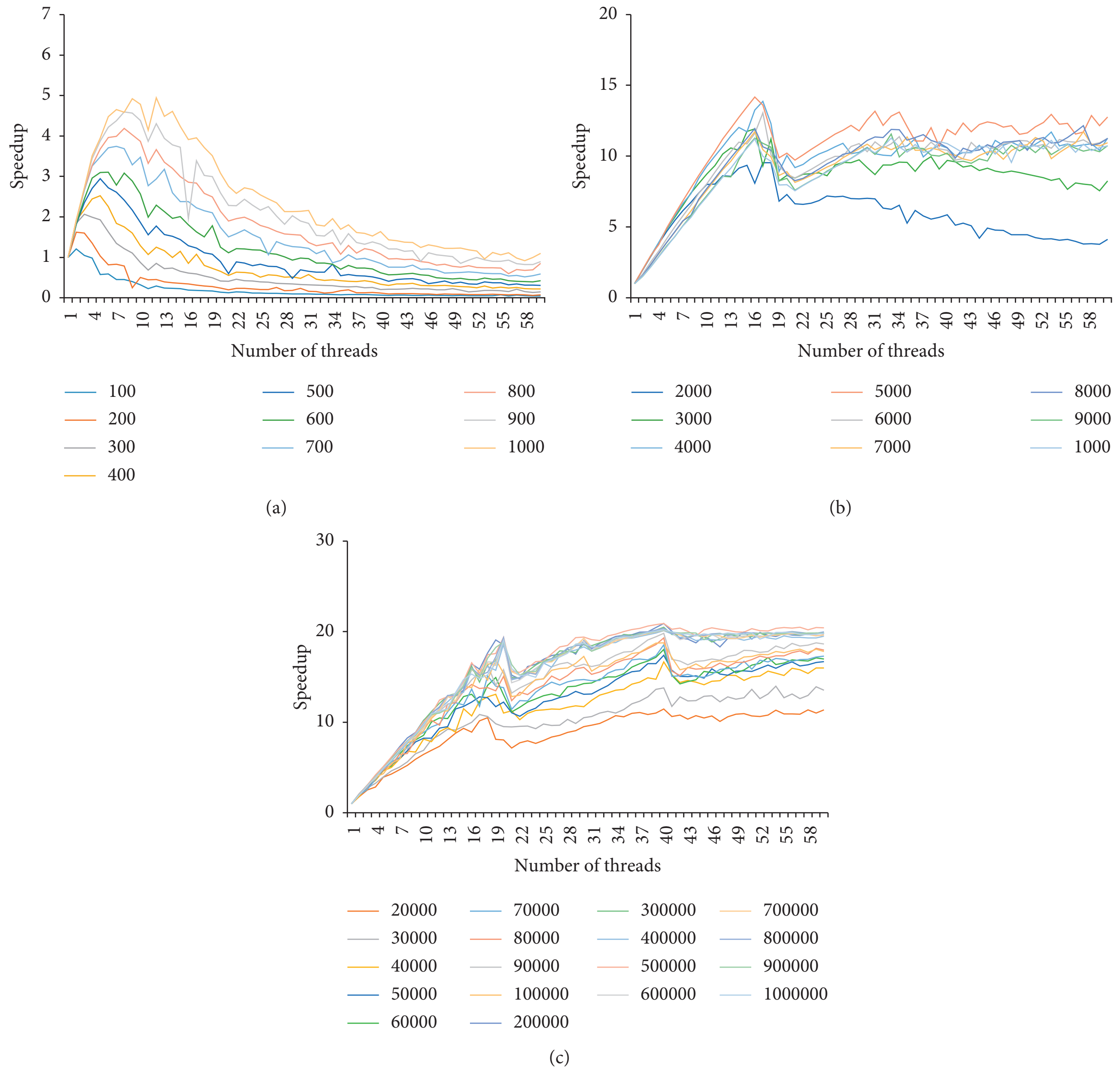

FIGURE 2: Performance with different number of threads. (a) Problem size from 100 to 1000, (b) problem size from 2000 to 10000, and (c) problem size from 20000 to 1000000.

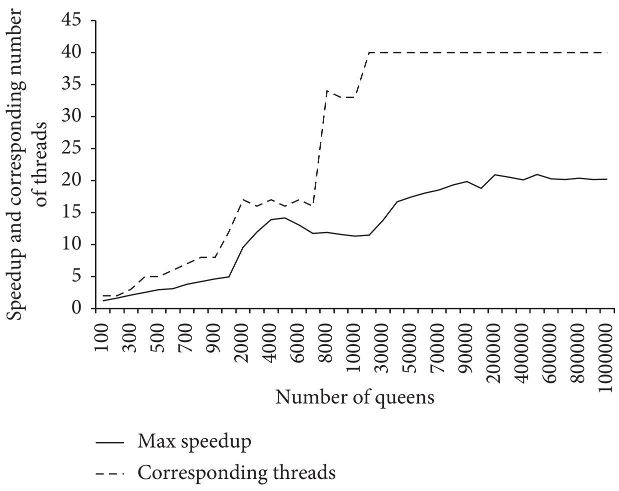

Figure 3: Best number of threads.

```
    Input: $N, NQ$
    Output: $m\_sum$
(1) begin 2
    int∗ m_NQ;/∗ array name ∗/
(3)   int m_N;/∗ array length ∗/
(4)   int m_sum;/∗ result ∗/
(5)   **Function**NQClass (int NQ[], int N):
(6)     m_NQ ⟵ NQ
(7)     m_N ⟵ N
(8)     m_sum ⟵ 0
(9)   **End Function**
(10)  **Function** NQClass (NQClass & x,split)
(11)    m_NQ ⟵ x.m_NQ
(12)    m_N ⟵ x.m_N
(13)    m_sum ⟵ 0
(14)  **End Function**
(15)  **Function** SubTask (int $i$)
(16)    conflict ⟵ 0;
(17)    **For** int $j = i + 1$; $j < N$; $++j$/∗ for every index after $i$ ∗/
(18)  **do**
(19)    **if** $NQ_i == NQ_j$ or $|NQ_j - NQ_i| == j-i$
    **then**
(20)        conflict++
(21)    **end**
(22)  **end**
(23)  **End Function**
(24)  **Function** operator (const blocked_range <int> & $r$):
(25)    int end ⟵ r.end()
(26)    **For** int $i$ = r.begin(); $i \neq$ end; $++i$) **do**
(27)      $m\_sum+ = \mathrm{SubTask}(m\_NQ, m\_N, i)$
(28)    **end**
(29)  **End Function**
(30)  **Function** join (const NQClass & y):
(31)    m_sum + = y.m_sum
(32)  **End Function**
(33) **end**
```

Algorithm 4: Class NQClass.

```
    Input: $N, NQ$
    Output: conflicts
(1) NQClass my_class (NQ, N)
(2) tbb:parallel_reduce(blocked_range < int > (0,N), my_class)
(3) conflicts ⟵ my_class.m_sum
```

Algorithm 5: Calculation of conflicts by using TBB Library.

```
     Input: $N, NQ$
     Output: conflicts
(1)  start ⟵ 0
(2)  end ⟵ N-2
(3)  if end–start < threshold;//calculate small task directly
(4)    then
(5)      For $i$ = start; $i$ ¡ = end; ++$i$ do
(6)        For $j = i + 1$; $j$ ¡ $N$; ++$j$ do
(7)          if $NQ_i == NQ_j$ or $|NQ_j - NQ_i|$ = = j–i
     then
(8)            conflicts++
(9)        end
(10)     end
(11)   end
(12) else
(13)   middle ⟵ (start + end)/2; //large task need to be splited
(14)   leftTask ⟵ new CountTask(start, middle); //generate sub_task_1
(15)   rightTask ⟵ new CountTask(middle+1,end); //generate sub_task_2
(16)   leftTask.fork(); //submit sub_task_1
(17)   rightTask.fork(); //submit sub_task_2
(18)   leftResult ⟵ leftTask.join(); //wait for sub_task_1
(19)   rightResult ⟵ rightTask.join(); //wait for sub_task_2
(20)   conflicts ⟵ leftResult + rightResult; //merge subtask results
(21) end
```

Algorithm 6: Calculation of conflicts with the Fork_Join framework.

computing power, which makes the performance of this scheme lower than that of the coarse-grained scheme by about two orders of magnitude. Since this huge performance difference cannot be compensated by thread task merging, we gave up this scheme when the number of queens reached 50000.

2.7. GPU Coarse-Grained Scheme. Each GPU thread corresponds to a queen's location to calculate whether this location conflicts with the queen behind the location and to accumulate the number of conflicts into the counter in the global memory of GPU with atomic operations. Algorithm 10 describes the task of one thread in GPU; it has a time complexity of $O(N)$.

The GPU kernel is launched with the following parameters:
Kerel $\ll <(N + \text{block\_size} - 1)/\text{block\_size}, \text{block\_size} \gg >$.
With Nvidia Tesla K80 [28], this scheme can theoretically be used to calculate the number of conflicts for $2^{31} - 1$ queens at maximum. Since only the candidate solution array needs to be transferred to the GPU, the data transmission burden of this scheme is very small.

2.8. GPU CDP-Based Scheme. The coarse-grained GPU scheme is relatively simple, but there are problems with unbalanced tasks between threads. For example, for queens number 2000, thread_0 has to check 1999 pairs of queen conflicts, while thread_1998 only checks whether 1 pair of queens conflict; therefore, the task amount of those two threads is 1999 times different. For larger N values, the issue is even more serious. To balance the amount of tasks between multiple threads, we set a threshold for using CDP (CUDA dynamic parallelism). CUDA dynamic parallelism technology appears for the first time in NVIDIA devices with a computing capability of 3.5 or higher. It empowers GPU kernels to launch nested subkernels by themselves, without the participation of the CPU, thereby avoiding the communication cost between the CPU and GPU. When a thread in the kernel is responsible for more comparisons between the two queens than the threshold, this thread uses dynamic parallel technology to launch subkernels, divides its task into more fine-grained

```
    Input: $N$
    Output: Pairs[ ]
(1) index ⟵ 0;
(2) For $p_i = 0$; $p_i \le N-2$; $p_i$++ do
(3)     For $p_j = p_i+1$; $p_j \le N-1$; $p_j$++ do
(4)         $\text{Pairs}_{\text{index}}$.x1 ⟵ $p_i$;
(5)         $\text{Pairs}_{\text{index}}$.x2 ⟵ $p_j$;
(6)         index++;
(7)     end
(8) end
```

Algorithm 7: Generation of subcript array in CPU.

```
    Input: $N, NQ$, Pairs
    Output: conflicts
(1) tid ⟵ global thread id in GPU Kernel
(2) $X_i$ ⟵ $\text{Pairs}_{tid}$.x1
(3) $X_j$ ⟵ $\text{Pairs}_{tid}$.x2
(4) $Y_i \longleftarrow NQ_{Xi}$
(5) $Y_j \longleftarrow NQ_{Xj}$
(6) if $Y_i == Y_j$ or $|Y_j - Y_i| == X_j - X_i$ then
(7)     atomicAdd(conflicts)
(8) end
```

Algorithm 8: Calculation of conflicts with fine-grained scheme 1 in GPU.

```
     Input: $N, NQ$
     Output: conflicts
 (1) tid ⟵ global thread id in Kernel;
 (2) NumofRound ⟵ $N$–1;
 (3) For iRound = 0; iRound ¡ N-1; iRound ++;/∗ find queens to be compared for current thread ∗/
 (4) do
 (5)     if tid - NumofRound¡ 0 then
 (6)         $X_i$ ⟵ iRound;
 (7)         $X_j$ ⟵ tid +1 +iRound;
 (8)         break;
 (9)     end
(10)     tid ⟵ tid - NumofRound;
(11)     NumofRound ⟵ NumofRound - 1;
(12) end
(13) $Y_i \longleftarrow NQ_{Xi}$ ;
(14) $Y_j \longleftarrow NQ_{Xj}$ ;
(15) if $Y_i == Y_j$ or $|Y_j - Y_i| == X_j - X_i$ then
(16)     atomicAdd(conflicts);
(17) end
```

Algorithm 9: Generate subscripts in GPU.

subtasks, and gives it to the subkernel to run. Algorithm 11 describes the process of deciding whether to call the subkernel according to the threshold, and Algorithm 12 describes the tasks of each subkernel. In theory, this scheme can alleviate the problem of unbalanced tasks between threads and also improve the parallelism. For the case of N = 2000 and threshold = 32, the task amount(pairs of queens to be checked) of thread_0 is 1999, which is greater than the threshold value of 32. This task is divided into 63 small subtasks with task amount not greater than the threshold. The number of subtasks can be determined

```
    Input: $N, NQ$
    Output: conflicts
(1) $X_i \longleftarrow$ global thread id in Kernel;
(2) $Y_i \longleftarrow NQ_{Xi}$ ;
(3) for $X_j = X_i + 1$; $X_j$ <N; ++ $X_j$ ;/* for index after $X_i$ */
(4) do
(5)     $Y_j \longleftarrow NQ_{Xj}$;
(6)     if $Y_i == Y_j$ or $|Y_j - Y_i| == X_j - X_i$ then
(7)         atomicAdd (conflicts);
(8)     end
(9) end
```

Algorithm 10: Calculation of conflicts with the coarse-grained scheme in GPU.

```
    Input: $N, NQ$
    Output: conflicts
(1) tid ⟵ global thread id in Kernel;
(2) num_tasks ⟵ N–tid -1;/* Get task amount of current thread by thread ID */
(3) if num_tasks $\leq$ threshold then
(4)  call Algorithm 10;/* For smaller tasks, calculate the results directly */
(5) else
(6)  num_subtasks = (num_tasks + threshold -1)/threshold; (Algorithm 11)
(7)  call Algorithm 12;/* Using dynamic parallel to call subkernel */
(8) end
```

Algorithm 11: CUDA dynamic parallelism scheme.

```
     Input: $N, NQ$
     Output: conflicts
 (1) sub_tid ⟵ global thread id in subkernel
 (2) $Y_i \longleftarrow NQ_{Xi}$
 (3) start $\longleftarrow X_i + 1 +$ threshold * sub_tid
 (4) end $\longleftarrow X_i + 1 +$ threshold * (sub_tid +1)
 (5) for $X_j$ = start; $X_j$ < end and $X_j$ < N; $X_j$ ++ do
 (6)     $Y_j \longleftarrow NQ_{Xj}$
 (7)     if $Y_i == Y_j$ or $|Y_j - Y_i| == X_j - X_i$ then
 (8)         atomicAdd (conflicts)
 (9)     end
(10) end
```

Algorithm 12: Task of the subkernel.

by the following formula: num_subtasks = ((num_tasks + threshold − 1)/threshold) = ((1999 + 32 − 1) /32) = 63 Each subtask is completed by a thread in the subkernel, so the launch parameters of the subkernel are as follows: Sub_Kerel ≪ < ((num_subtasks + sub_block_size − 1)/sub_block_size, sub_block_size ≫ >). The parent grid and the subkernel have their exclusive local memory and shared memory space, so the parent kernel should pass data to the child kernel by passing values or global memory space pointers instead of pointers of local memory or shared memory space.

3. Experiment

3.1. Test Platform.

3.1. Test Platform. All trials were performed on an HP Proliant DL580 Gen9 server with a Tesla K80. The configuration of the experimental platform is shown in Table 2.

With 2 Xeon E7-4820 v4 CPU, our experimental platform has 20 physical cores, which can be virtualized into 40 logical cores through Intel Hyper-Threading Technology and run 40 threads at the same time. Nvidia Tesla K80 has up to 2.91 Teraflops of double-precision performance and

Table 2: Experiment configuration.

	CPU	GPU
Processors	Intel(R) Xeon(R)	Nvidia Tesla K80
Chips	E7-4820 v4 * 2	GK210 * 2
Cores/chip	10 cores	2496 CUDA cores
Frequency	2.0 GHz	560 MHz 824 MHz
Memory	128 GB	12 GB *2
Compiler version	g++ 5.4.0	CUDA 8.0
OS	Ubuntu16.04	Ubuntu16.04

480 GB/sec memory bandwidth. Most of the experimental data used in charts below were averaged over 100 runs, and a few very time-consuming experimental data use the average value of 10 runs. We use the high-precision std: : chrono library provided in C ++11 standard for timing and use microseconds as the timing unit for the small-scale problem ($N \leq 900$) and milliseconds for the large-scale problem ($N > 900$). CUDA function cudaEventSynchronize is used for synchronization between GPU and CPU.

Schemes based on CPU are implemented with C++ and Java, and schemes based on GPU are implemented with CUDA-C. We set the block size of GPU kernels to 512 based on experience. For GK210, the maximum number of concurrent threads in each SM is 2048, and the block size we set can ensure that there are 4 blocks in each SM. Because the number of subkernel threads caused by CUDA dynamic parallelism is often in the order of tens and hundreds, we set a smaller block size (32) for the subkernel. For fairness and portability on different hardware, all codes are compiled with the default optimization option.

Java virtual machine supports hotspot detection technology, which can analyze hotspot code and optimize it automatically. We tried $C1$ and $C2$ compiler with the (−server/−client) option in the compilation phase and forced JIT mode with the -Xcomp option in the runtime phase. The results show that the best performance can be achieved by default compiling and running configuration .

3.2. Performance Comparison. As shown in Figure 4, the rank of performance of seven schemes is a coarse-grained GPU scheme, multithreaded CPU schemes (including an adaptive scheme and TBB scheme), Fork-Join scheme, single-threaded CPU scheme, and fine-grained GPU schemes 1 and 2. The dynamic parallelism scheme will be discussed in Section 3.3 separately.

Because of the cost of thread startup and management, the performance of the CPU-adaptive multithreaded scheme and TBB scheme is lower than that of the single-threaded scheme with small problem sizes. However, as the problem size increases to 300, these two schemes keep their advantages over other schemes until the problem size is more than 3000, which is replaced by the GPU coarse-grained scheme.

Fork-Join scheme has more extreme characteristics: when the problem size is less than 2000, its performance is even lower than that of the single-threaded scheme; when the problem size reaches 50000, its performance exceeds that of the multithreaded scheme and TBB scheme. The highest speedup of multithreaded and TBB schemes is 22.04 and 20.71, while the Fork-Join scheme achieved a maximum speedup of 29.18. Considering that 40 logical cores of our experimental platform are virtualized on 20 physical cores by Intel Hyper-Threading Technology, this scheme has achieved a high CPU resource utilization.

The performance of fine-grained scheme 1 is lower than that of the CPU scheme because of the large memory usage, high transmission cost, and the number of tasks per thread being too small to offset the overhead caused by thread management.

Fine-grained scheme 2 gave the task of calculating the subscripts of the queens to be compared to the GPU to avoid a large amount of data transmission. However, the SIMT architecture of the GPU is suitable for executing a code with large amount of tasks and simple control structure. While calculating subscripts of the queens, we use loop and branch structure, which causes GPU thread divergence. In the most extreme cases, 32 threads in a warp will execute in sequence and seriously reduce the execution efficiency. The experimental results show that, in the heterogeneous architecture of CPU + GPU, the optimization scheme must comprehensively measure various factors, such as thread parallelism, data transmission throughput, SM (Streaming Multiprocessor) core utilization, and load balancing among multiple SMs. Only one factor of increasing parallelism does not necessarily lead to performance improvement.

When the problem size $N < 700$, the performance of the coarse-grained GPU scheme is lower than that of the CPU single-threaded counterpart. The reason is that, in the case of problems with small sizes, only a small number of threads participate in the calculation, the utilization of GPU cores is low, and the overhead caused by the GPU thread startup and data transmission covers the gain brought by computational parallelism. When $N \geq 700$, the advantages brought by the massive parallelism of the GPUs make the speedup to the single-threaded scheme continue to rise. When the size of the problem reaches 400,000, the speedup is stable at approximately 300, ten times more than that of the CPU multithreaded scheme. These phenomena can also be obtained by observing the speedup changes of these schemes under different problem sizes, which is described in a logarithmic form in Figure 5.

3.3. Performance of the GPU CDP-Based Scheme. NVIDIA Tesla K80 has a computing capability of 3.7 and supports dynamic parallelism. Function cudaDeviceSetLimit cudaLimitDevRuntimeSyncDepth,MAX DEPTH is used to set the depth of the dynamic parallel stack, and the maximum depth is 24. If the program's recursive call depth of dynamic parallel exceeds this limit, no error will be reported, but the result returned from GPU is wrong.

In our scheme, for threads whose task amount exceeds the threshold, the dynamic parallel is only triggered once,

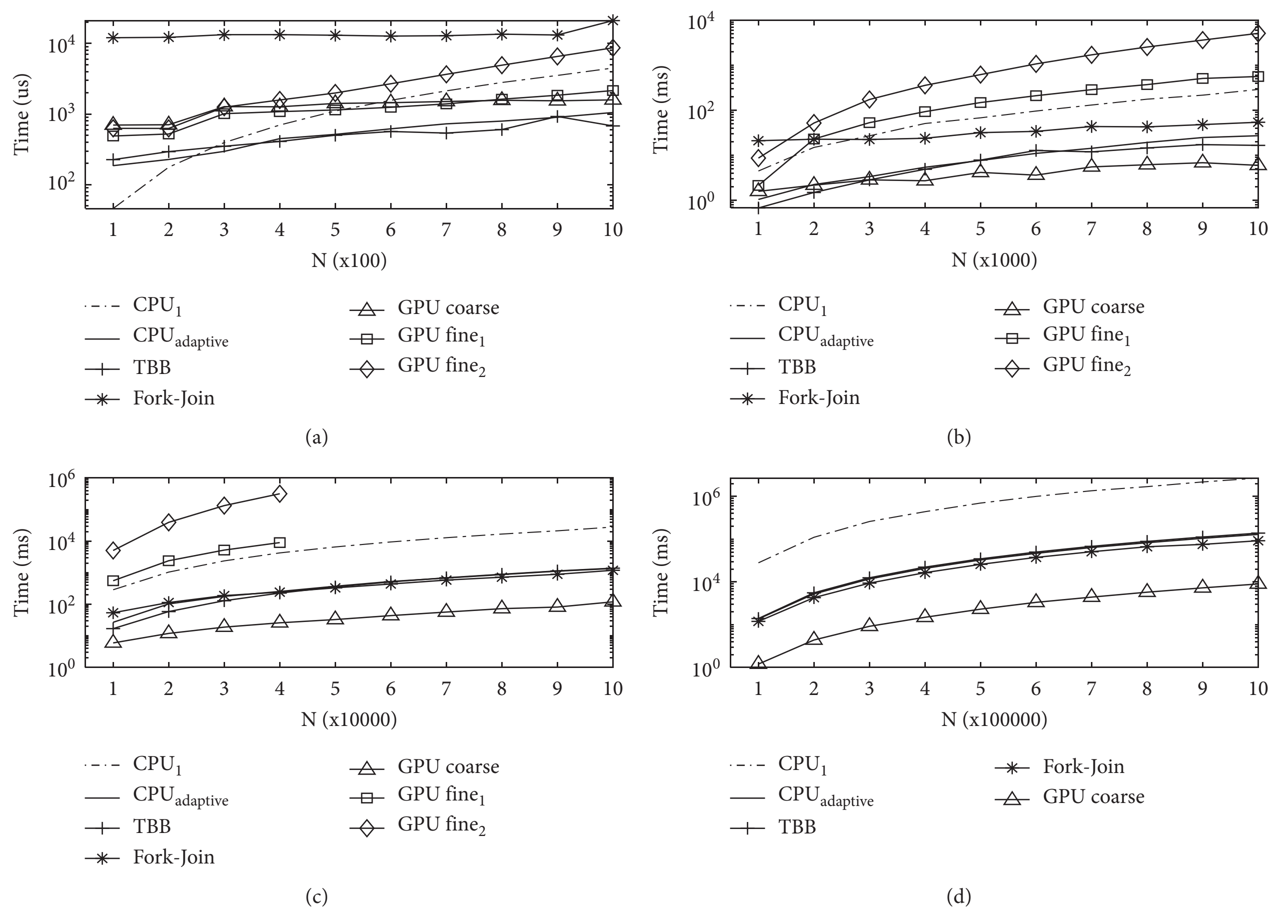

Figure 4: Performance comparison of Sections 2.1–2.7 in different problem sizes. (a) Microsecond is used as the time unit; (b)–(d) millisecond is used as the time unit; the Y-axis uses logarithmic coordinates.

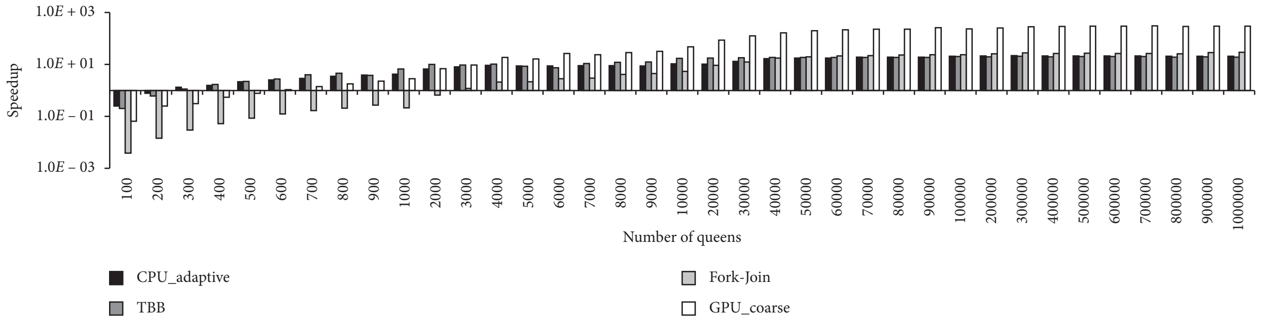

Figure 5: Speedups of schemes in Sections 2.1–2.7.

and the recursion depth of dynamic parallel is 1. The task is divided into several subtasks whose task amount is less than the threshold. Because the dynamic parallel scheme is very sensitive to the threshold parameters, we discuss the performance of this scheme separately in this section.

Tables 3 and 4 record the running time of the dynamic parallelism scheme when the threshold is set to 1 k, 2 k, 4 k, 8 k, 16 k, 32 k, 64 k, and 128 k. The first column is the number of queens. The second column is the running time of the coarse-grained GPU scheme, which is used here for comparison. The remaining columns are the running time of the dynamic parallel scheme with different thresholds. The time unit used here is millisecond.

The experimental results show that no matter how large the threshold is, the performance will drop sharply as long as the dynamic parallelism is triggered. This experiment shows that CDP is not suitable for the acceleration of the evaluate function of the N-Queens problem. We believe that, for the larger N-Queens problem, the coarse-grained GPU scheme has started enough threads and reached a high SM occupation. If one GPU thread uses dynamic parallelism to launch new subkernels, it will

Table 3: Performance of the CDP-based scheme. Threshold from 1 k to 8 k.

N	Coarse-grained scheme	1024	2048	4096	8196
1000	1	1	1	2	2
2000	2	242	2	2	2
3000	3	234	447	3	3
4000	4	665	463	3	4
5000	4	1168	1218	856	3
6000	5	1753	2171	881	4
7000	6	2394	3078	2235	5
8000	6	2949	4160	4049	6
9000	7	3592	5098	5746	1699

Table 4: Performance of the CDP-based scheme. Threshold from 16 k to 128 k.

N	Coarse-grained scheme	16 k	32 k	64 k	128 k
10000	7	8	6	6	6
20000	14	12891	14	14	12
30000	21	81920	16	18	22
40000	29	155067	73637	24	24
50000	37	227734	211633	30	28
60000	52	301992	350017	52	38
70000	71	379358	490801	73664	57
80000	87	457052	639331	340804	72
90000	91	538742	777274	613907	84

not reduce the total workload (queens comparison times), but instead increase the number of extra thread startup and the management workload, resulting in performance degradation.

CDP is very suitable for writing recursive patterns to implement divide and conquer or backtracking algorithm. The advantage of this technology is to deal with irregular tasks, such as searching in the unbalanced tree of N-Queens problem variant 3. However, to evaluate the candidate solution of the N-Queens problem with a fixed size is a regular workload and its calculation amount can be predicted in advance, and the overhead caused by dynamic subkernel launches outweighs the benefits of the improved load balance yielded by CDP.

3.4. Stability Analysis of the Coarse-Grained Scheme. Statistics show that a random candidate solution contains a number of conflicts of approximately 2/3 of the length of the solution. In the evolutionary process of heuristic algorithms, candidate solution will continue to evolve in the optimal direction, and the number of conflicts included in the candidate solution will gradually decrease until a valid solution with zero conflict appears. This process leads to a reduction in the number of atomic operations in the evaluation process of the candidate solution, which theoretically shortens the running time of the coarse-grained GPU scheme.

To observe the effect of the reduction of the conflict number on the performance of the coarse-grained GPU scheme, we test the performance of the coarse-grained GPU scheme with a random solution set and valid solution set with length ranging from 100 to 1 million. We shuffle the sequence from 1 to N to construct the random candidate solution set and use the Hoffman construction method to construct the valid solution set.

As shown in Figure 6, as the length of the solution increases, the performance difference on the two datasets gradually decreases. The reason is that as the length of the solution increases, the number of threads and the amount of computation gradually increase, and the delay caused by the atomic operation in GPU global memory has more chances to be hidden by other threads/warp running. Compared with the valid solution set, the performance of the coarse-grained GPU scheme on the random candidate solution set is reduced by 0.95% on average. The fluctuation of about 1% indicates that the coarse-grained GPU scheme has strong stability in different datasets.

4. Application of the GPU Coarse-Grained Scheme to Simulated Annealing

In order to verify the effectiveness of our scheme, we apply the coarse-grained GPU scheme to simulated annealing to solve N-Queens problem variant 2. We keep the parameters and the experimental platform intact and only replace the evaluation function from the CPU single-threaded scheme to the coarse-grained GPU scheme.

As can be seen from Figure 7, because the evaluation function takes a very high proportion of time in the whole

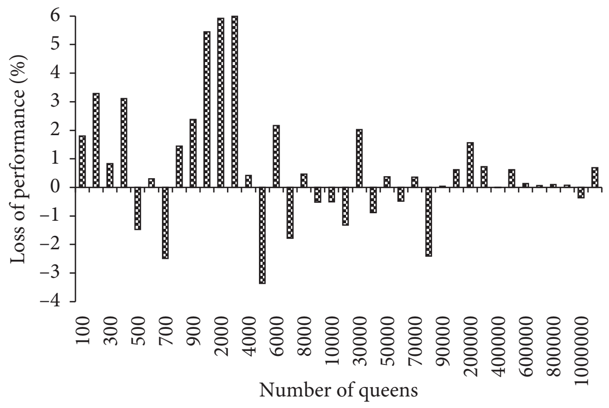

Figure 6: Performance loss of coarse-grained GPU scheme in the worst case.

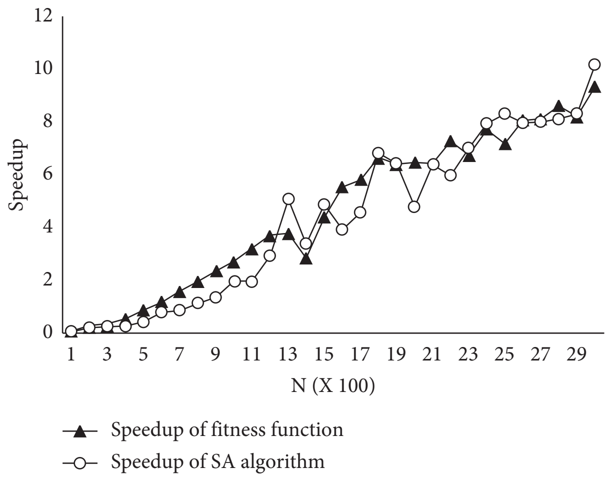

Figure 7: Speedup of the GPU scheme in SA algorithm.

simulated annealing algorithm, the performance gain of fitness function brought by GPU parallelism directly improves the performance of the SA algorithm. Taking into account the experimental errors and that simulated annealing is a probabilistic technique, we think that the acceleration of fitness function is directly reflected in the SA algorithm\enleadertwodots.

5. Conclusions

Variant 2 of the N-Queens problem is a classical problem which has been proved to be NP-hard, so heuristic algorithms are often used to obtain valid solutions. At present, the parallelization of these methods is often at the algorithm level, such as dividing the large population into several small populations for evolution in parallel or mutating individuals in parallel. In this paper, we focus on how to improve the speed of the heuristic algorithms for solving variant 2 by accelerating the evaluation function.

Besides a CPU single-threaded serial scheme, three CPU multithreaded parallel schemes and four GPU parallel schemes are proposed to evaluate candidate solutions for the N-Queens problem. The performances of all schemes are measured under uniform experimental. In solving N-Queens problem variant 2 with the simulated annealing algorithm, the advantage of the coarse-grained GPU scheme has been proved. Usually, the evaluation function is the most time-consuming part of a heuristic algorithm, and we believe that our schemes based on CPU and GPU can improve the performance of all algorithms based on search and evaluation in solving N-Queens problem variant 2 without changing the algorithm process and parameters. These algorithms include simulated annealing, genetic algorithm, chemical reaction optimization, etc. Users can choose the appropriate scheme according to their hardware devices to speed up their computing process. Our scheme does not conflict with the parallel methods at the algorithm level; they can be used together. For example, in the case of GPU hardware, replacing the fitness function in the island model of parallel genetic algorithm with the GPU scheme proposed in this paper can further shorten the execution time.

The performance of the coarse-grained GPU scheme can be further improved by the following means, which is also our next work:

(1) In the current GPU coarse-grained GPU scheme, device memory is allocated and freed for each evaluation function call. Performance can be improved theoretically if device memory is reused in multiple evaluation function calls. Data transmission from CPU to GPU and computation in kernel can be parallel by using CUDA multistream technology.

(2) Some CUDA kernel configuration parameters can be further optimized. We plan to use NVidia NVVP [29] to read the hardware counters in the GPU to analyze the microperformance bottleneck of the program and to further improve the performance of GPU schemes by optimizing parameters, such as the block size and L1D/Share memory settings. For the dynamic parallel scheme, we plan to use bypass technology to cancel some subkernel launch random in order to reduce the thread management burden and improve performance.

(3) This paper focuses on using thread-level parallel technology to improve the efficiency of the evaluation algorithm. Instruction-level parallelism is also an important optimization method. In the next step, we plan to combine these two methods to further improve the performance of the algorithm.

Data Availability

The experimental code, script, and result used to support the findings of this study are available at https://github.com/grasshoper97/NQueens.git.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this manuscript.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (61672123).

References

[1] F. W. M. Bezzel, "Proposal of eight queens problem," *Berliner Schachzeitung*, vol. 3, p. 363, 1848.

[2] J. Bell and B. Stevens, "A survey of known results and research areas for," *Discrete Mathematics*, vol. 309, no. 1, pp. 1–31, 2009.

[3] E. J. Hoffman, J. C. Loessi, and R. C. Moore, "Constructions for the solution of the m queens problem," *Mathematics Magazine*, vol. 42, no. 2, pp. 66–72, 1969.

[4] J. Somers, "The n-queens problem - a study in optimization. [EB/OL]," 2019, http://jsomers.com/nqueendemo/nqueens.

[5] K. Kise, T. Katagiri, H. Honda, and T. Yuba, "Solving the 24-queens problem using Mpi on a Pc cluster," Technical Report, Graduate School of Information Systems, The University of Electro-Communications, Tokyo, Japan, 2004.

[6] D. Caromel, A. Di Costanzo, and C. Mathieu, "Peer-to-peer for computational grids: mixing clusters and desktop machines," *Parallel Computing*, vol. 33, no. 4, pp. 275–288, 2007.

[7] T. B. Preußer, B. Nagel, and R. G. Spallek, "Putting queens in carry chains," Technical Report, Technische Universität Dresden, Dresden, Germany, 2009.

[8] T. Carneiro, A. E. Muritiba, M. Negreiros, and G. A. Lima De Campos, "A new parallel schema for branch-and-bound algorithms using gpgpu," in *Proceedings of the 2011 23rd International Symposium on Computer Architecture and High Performance Computing*, pp. 41–47, Washington, DC, USA, October 2011.

[9] L. Li, H. Liu, H. Wang, T. Liu, and W. Li, "A parallel algorithm for game tree search using gpgpu," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 8, pp. 2114–2127, 2015.

[10] F. Feinbube, B. Rabe, M. Von Löwis, and A. Polze, "Nqueens on cuda: optimization issues," in *Proceedings of the 2010 Ninth International Symposium on Parallel and Distributed Computing*, Istanbul, Turkey, July 2010.

[11] D. Amrasinghe, "N-queens problem with gpgpu (presentation)," Technical Report, University of North Texas, Denton, TX, USA, 2007.

[12] V. Pamplona, "N-queens problem: a comparison between cpu and gpu using c++ and cuda (presentation)," Technical Report, Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil, 2008.

[13] T. Zhang, W. Shu, and M. Y. Wu, "Optimization of N-queens solvers on graphics processors," in *Proceedings of the 9th International Conference on Advanced Parallel Processing Technologies*, Berlin, Heidelberg, August 2011.

[14] K. Thouti and S. R. Sathe, "Solving N-queens problem on gpu architecture using opencl with special reference to synchronization issues," in *Proceedings of the 2012 2nd IEEE International Conference on Parallel, Distributed and Grid Computing*, Himachal Pradesh, India, December 2012.

[15] M. Plauth, F. Feinbube, F. Schlegel, and A. Polze, "Using dynamic parallelism for fine-grained, irregular workloads: a case study of the n-queens problem," in *Proceedings of the 2015 Third International Symposium on Computing and Networking (CANDAR)*, Hokkaido, Japan, December 2015.

[16] T. Carneiro Pessoa, J. Gmys, F. H. De Carvalho Júnior, N. Melab, and D. Tuyttens, "Gpu-accelerated backtracking using cuda dynamic parallelism," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 9, Article ID e4374, 2018.

[17] T. Carneiro, J. Gmys, N. Melab, F. H. De Carvalho Junior, P. P. Rebouças Filho, and D. Tuyttens, "Dynamic configuration of cuda runtime variables for cdp-based divide-and-conquer algorithms," in *High Performance Computing for Computational Science–VECPAR 2018*Springer International Publishing, Berlin, Germany, 2019.

[18] X. Hu, R. C. Eberhart, and Y. Shi, "Swarm intelligence for permutation optimization: a case study of n-queens problem," in *Proceedings of the 2003 IEEE Swarm Intelligence Symposium. SIS'03 (Cat. No.03EX706)*, pp. 243–246, Indianapolis, IN. USA, April 2003.

[19] P. Jafarzadeh and V. M. Saffarzadeh, "A hybrid approach using particle swarm optimization and parallel simulated annealing: a case study of n-queens problem," *International Journal of Computer and Electrical Engineering*, vol. 6, no. 3, pp. 231–235, 2014.

[20] X. Y. M. Guang-Yong and Y. Ying, "On chemical reaction optimization for eight queens problem," *Journal of Hengyang Normal University*, vol. 35, no. 3, pp. 109–114, 2014.

[21] H. Nengfa and K. Lishan, "A global optimization algorithms with integer encoding," *Journal of Hubei University*, vol. 24, no. 2, pp. 123–126, 2002.

[22] C. Y. Buzhong and W. Yibin, "On-chip multi-core parallel hybrid genetic algorithm for solving n-queens problem," *Computer Engineering*, vol. 41, no. 7, pp. 123–126, 2015.

[23] A. M. Turky and M. S. Ahmad, "Using genetic algorithm for solving n-queens problem," in *Proceedings of the 2010 International Symposium on Information Technology, 2010 International Symposium on Information Technology*, Kuala Lumpur, Malaysia, June 2010.

[24] K. Wang, Z. Ji, and Y. Zhou, "A parallel genetic algorithm based on Mpi for N-queen," in *Proceedings of the 2017 2nd International Conference on Control, Automation and Artificial Intelligence (CAAI 2017)*, Sanya, China, June 2017.

[25] C. Jianli, C. Zhikui, W. Yuxin, and G. He, "Parallel genetic algorithm for N-Queens problem based on message passing interface-compute unified device architecture," *Computational Intelligence*, vol. 36, no. 4, pp. 1621–1637, 2020.

[26] S. Kirkpatrick, D. Jr, and M. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, pp. 671–680, 1983.

[27] NVIDIA, CUDAC Programming Guide, NVIDIA, 2016.

[28] NVIDIA, NVIDIA TESLA K80, *The World's Fastest GPU Accelerator*, NVIDIA, Santa Clara, CL, USA, 2014.

[29] NVIDIA, CUDA Profiler Users Guide, NVIDIA, 2016.