Hindawi
Wireless Communications and Mobile Computing
Volume 2021, Article ID 1999154, 12 pages
https://doi.org/10.1155/2021/1999154

Research Article

Parallel Differential Evolutionary Particle Filtering Algorithm Based on the CUDA Unfolding Cycle

Kaijie Huang and Jie Cao

Lanzhou University of Technology, Lanzhou, China

Correspondence should be addressed to Kaijie Huang; h18893471259@gmail.com

Received 13 August 2021; Revised 31 August 2021; Accepted 9 September 2021; Published 15 October 2021

Academic Editor: Rajesh Kaluri

Copyright © 2021 Kaijie Huang and Jie Cao. 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.

Aiming at the problem of low statute efficiency of prefix sum execution during the execution of the parallel differential evolutionary particle filtering algorithm, a filtering algorithm based on the CUDA unfolding cyclic prefix sum is proposed to remove the thread differentiation and thread idleness existing in the parallel prefix sum by unfolding the cyclic method and unfolding the thread bundle method, optimize the cycle, and improve the prefix sum execution efficiency. By introducing the parallel strategy, the differential evolutionary particle filtering algorithm is implemented in parallel and executed on the GPU side using the improved prefix sum computation during the algorithm update. Through big data analysis, the results show that this parallel differential evolutionary particle filtering algorithm with the improved prefix sum statute can effectively improve differential evolutionary particle filtering for nonlinear system states and real-time performance in heterogeneous parallel processing systems.

1. Introduction

Particle filtering is a sequential Monte Carlo method that employs particles to approximate the posterior probability density distribution. In [1], the multi-intelligent coevolution mechanism is introduced into particle filtering, and the resampling process is realized by the competition, crossover, mutation, and self-learning among particles, which effectively solve the problem of particle degradation and particle scarcity. Literature [2] compared the filtering accuracy of particle filtering under different search strategies, and the accuracy of the differential evolutionary particle filtering algorithm was improved, but the computational complexity was increased. To address the computational complexity problem, literature [3–5] proposed a GPU-based particle filtering parallel algorithm, which effectively combines the traditional particle filtering algorithm with GPU to make full use of the performance of GPU parallel computing and accelerate the computational speed of the particle filtering algorithm. Literature [6, 7] proposed a GPU-based parallel optimization design and implementation of particle filtering to improve the computational speed of the tracking algorithm. Literature [8–10] designed and implemented a parallel particle swarm optimization algorithm based on CUDA, which uses a large number of GPU threads to accelerate the convergence speed of the whole particle swarm. Parallel statute algorithms are used in the abovementioned literature for parallel particle filtering algorithms to simplify thread operations. Prefixes and algorithms are an important primitive for parallel algorithm programming and are utilized as basic modules for many different algorithms. Compared to serial algorithms, CUDA-based parallel algorithms execute single instruction multithreaded commands, which can perform more operations and improve the efficiency of algorithm execution. However, due to the execution mode and memory access mode of the prefix sum algorithm [11–13], the execution process is prone to thread division and memory access conflict phenomena, which cannot effectively utilize the hardware resources of GPU. Prefix summation contains a large number of repetitive operations, which are simple but inefficient. Segmented prefix summation avoids thread repetition but suffers from serious memory access problems, making the utilization of GPU hardware resources low. Literature [14] introduces additional instructions and

demonstrates their application in the construction of efficient parallel algorithm primitives, such as prefix sums and segmented binary prefix sums. In literature [15], researchers used parallel segmented prefixes to construct data processing and optimize them to improve the overall performance of the algorithm. In literature [16], researchers used GPUs and the practical parallel particle swarm well to solve the problem of singular facility locations, demonstrating that particle swarm optimization is a flexible optimization technique. In literature [17], several tree data structures are studied for the prefix sum problem, providing a variety of practical solutions, all of which obtain a good speedup factor.

To address the problem of thread differentiation in the execution of the differential evolutionary particle filtering parallel algorithm, based on CUDA architecture, this paper proposes a differential evolutionary particle filtering algorithm based on unfolding cyclic prefixes and optimization to remove thread differentiation and reduce the lag caused by judgment and branch prediction, which makes the particle filtering algorithm gradually improve the computational performance.

2. Differential Evolutionary Particle Filtering Algorithm

Differential evolutionary algorithm (DE) is a stochastic parallel direct search algorithm, whose basic idea is to start from a certain randomly generated initial population, iterate continuously according to certain operation rules, and according to the fitness value of each individual, keep the good individuals and eliminate the inferior ones, and guide the search process to approach the optimal solution. The algorithm has the advantages of simple structure, easy implementation, no need for gradient information, fewer parameters, etc., and has a variety of different search strategies.

The calculation process of the DE-PF algorithm in this paper is as follows.

Step 1. For the initialization step, sampling is performed at time $k=0$. The resulting N particles $\{x_0^i\}_{i=1}^N$ are used as initial samples, and the distribution of the initial samples is $x_0^i \sim p(x_0)$. All particles have the same initial weights $w_0^i = 1/N$. Repeat iterations for $T = 1, 2, 3, \cdots, N$.

Step 2. For the prediction step, set$k = k+1$, sample particle $\{x_k^i\}_{i=1}^N$ at the current moment through the state transfer model, and calculate the current measure $\{y_k^i\}_{i=1}^N$.

Step 3. The weights are calculated and normalized, and after receiving the measurements in Step 2, each particle needs to update the weights according to the likelihood function $p(y_T | x_T^i)$:

$$w_t^i = w_{t-1}^i \cdot p\left(y_t \middle| x_t^i\right). \quad (1)$$

The normalization process makes the sum of the particle weights equal to one, and the normalization process is expressed as

$$w_t^i = \frac{w_t^i}{\sum_N w_t^i}. \quad (2)$$

Step 4. For differential evolutionary resampling, we have the following:

(1) $g=1$. The initial particle of evolution $\{x_k^{g,i}\}_{i=1}^N = \{x_k^i\}_{i=1}^N$

(2) The variation operation is performed on the particle set $\{x_k^{g,i}\}_{i=1}^N$, and then the crossover operation is performed to obtain the candidate particle set $\{\tilde{x}_k^{g,i}\}_{i=1}^N$

(3) The fitness value of the candidate particle set $\{\tilde{x}_k^{g,i}\}_{i=1}^N$ is calculated, and the selection operation is performed, and the resulting particle set is$\{x_k^{g+1,i}\}_{i=1}^N$

(4) If $g < G_{\max}$ and $\sigma > \sigma_{\min}$, then set $g = g+1$; turn to Step 2; otherwise, go to the next step

Step 5. Arrange the particles in descending order.

Step 6. Count the number of times each particle is copied, except for its own.

Step 7. Calculate the weighted sum of the weights in Step 4, except for its own.

Step 8. Eliminate the small particles.

Step 9. For the state output step, the optimized set of particles is used as a sample of equal weights $\{x_k^i, w_k^i = N^{-1}\}_{i=1}^N$:

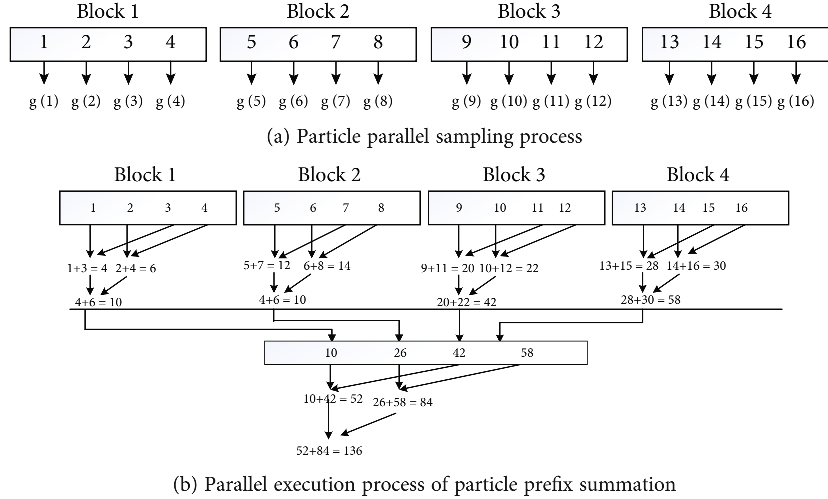
$$\text{Calculated state estimates}: x_k = \sum_{i=1}^N w_k^i, x_k^i. \quad (3)$$

3. Improved Parallel Prefix Sum

The parallel algorithm needs to calculate the cumulative distribution function (CDF) of the particles when performing the computation, which is a simple continuous prefix and operation described as follows:

$$y[n] = y[n-1] + x[n], \quad (4)$$

where$n = 0, 1, N-1$,$y[-1] = 0$, and$N$is the size of the data$y[n]$. The sequential computation is very straightforward and makes parallelization difficult due to the dependencies between output data. For small prefix sum problems, only one thread block is used and recursive multiplication is used to solve the problem. However, parallel particle filtering requires a longer computation of the prefix sum problem when the number of particles $N = 16$, and Figure 1 expresses the same operation on different particles, i.e., the parallel way of prefix summation.



(a) Particle parallel sampling process

(b) Parallel execution process of particle prefix summation

Figure 1: Parallel prefix sum based on unfolding loop improvement.

The parallel prefix sum can be understood as the parallelization of the process of summing all the numbers in an array. In general, the idea of parallelization is based on the binary statute of "trees," as shown in Figures 2 and 3. The implementation of parallel prefix summation can be divided into two types:

(1) Direct Prefix Sum. Elements are paired with their direct neighbors to find the sum

(2) Interleaved Prefix Sums. Elements are paired according to a given span

Based on the problem of idle threads in the parallel operation of the interleaved prefix sum algorithm, this paper proposes a spread-loop prefix sum method to reduce idle threads and improve the efficiency of prefix sum execution.

By assessing the interleaved prefix sum method, the initial value of stride is half of blockDim.x. When (tid < stride) and then executing subsequent instructions, it means that half of the threads in the first iteration are idle, which wastes GPU computing resources and targets a new problem: idle threads. The performance of the parallel algorithm can still be improved if all of them can be utilized, which is also pending the next step to be optimized and improved.

Expanding loops is a technique that is intended to optimize loops by reducing the frequency of branch occurrences and loop maintenance instructions. In a loop expansion, the body of the loop is written multiple times in the code, rather than just writing the body of the loop once and then using another loop to execute it repeatedly. Any closed loop can have its number of iterations reduced or removed altogether. The number of copies of the loop body is referred to as the loop expansion factor, and the number of iterations becomes the singular number of iterations divided by the loop expansion factor. In sequential arrays, loop expansion is the most efficient way to improve performance when the number of iterations of the loop is known before the loop is performed. Assuming a thread block length of 1024, the threads involved in the computation of the statute iterations at 512, 256, 128, and 64 are distributed in different thread bundles (since each warp can only have 32 threads executing simultaneously); then, there is an order of precedence in the SM execution of these thread bundles, so each step of the statute iteration needs to be synchronized within the block. Only when the statute iterates to 32, 16, 8, 4, and 2, the thread bundle execution they are in is not associated with other thread bundles and no interblock synchronization is needed, while there is implicit synchronization after each instruction in the process of thread bundles in SM, so the intrabundle synchronization problem can be solved, making the global array corresponding to the threads get updated in time without affecting the execution of the next instruction.

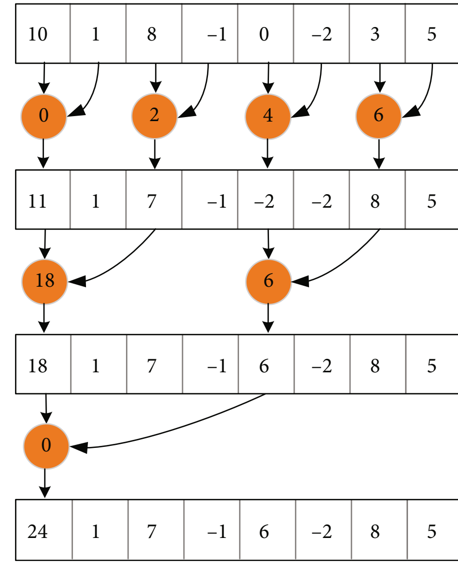


Figure 2: Direct prefix sum.

In the preceding prefix and computation, each thread block was responsible for one corresponding data block.

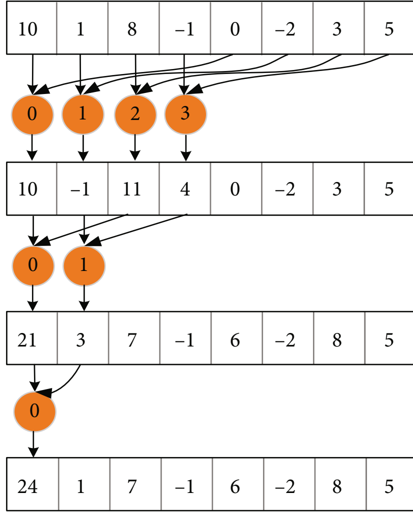


Figure 3: Interleaved prefix sum.

Table 1: Interleaving prefix and expanding loop (expanding factor is 2).

```
if (index + blockDim.x < N)d_data[index] + = d_data
[index + blockDim.x];
__syncthreads();
for (int strize = blockDim.x/2 ; strize > 0 ; strize>> = 1)
{if (tid < strize)
data[tid] + = data[tid + strize];
__syncthreads();
```

Now, each thread block is responsible for prefixing and calculating two data blocks, thus eliminating instruction consumption and increasing the scheduling of more independent instructions to improve performance. The following is a schematic diagram of the prefix sum with expansion factors of 2 and 4. There are three scales of expansion, 2, 4, and 8, where a block computes 2 blocks, 4 blocks, and 8 blocks of data, respectively, adds the adjacent data blocks to the data block corresponding to the current thread block, and then sums them, listed as Tables 1–3.

The parallel prefix and method algorithm strength is low, so the bottleneck in the system may be due to the scheduling instructions. The solution is to expand the for loop. __syncthreads is used for intrablock synchronization. In the statute kernel function, it is used to ensure that all threads in each round have written their local results to global memory before the thread moves to the next round. During the statute, the number of active threads decreases, and when there are less than 32 active threads, we will have only one warp. In a single warp, the execution of instructions follows the SIMD (single instruction multiple data) pattern; i.e., when there are less than 32 active threads, there is no need for synchronization control, and each instruction is followed by an implicit intrabundle synchronization process after each instruction. Therefore, it is necessary to solve the problem of loop control and thread synchronization when

Table 2: Interleaved prefix and loop expansion with a factor of 4.

```
if (index + 3 * blockDim.x < N)
{int a = d_data[index];
int a1 = d_data[index + blockDim.x];
int a2 = d_data[index + 2 * blockDim.x];
int a3 = d_data[index + 3 * blockDim.x];
d_data[index] = (a + a1 + a2 + a3);
```

Table 3: Interleaved prefix and thread bundle expansion.

```
if (tid < 32)
{volatile int * vmen = data;
vmen[tid] + = vmen[tid + 32];
vmen[tid] + = vmen[tid + 16];
vmen[tid] + = vmen[tid + 8];
vmen[tid] + = vmen[tid + 4];
vmen[tid] + = vmen[tid + 2];
vmen[tid] + = vmen[tid + 1];
```

Table 4: Interleaved prefixes and fully expanded.

```
if (index + 7 * blockDim.x < N)
int a = d_data[index];
int a1 = d_data[index + blockDim.x];
int a2 = d_data[index + 2 * blockDim.x];
int a3 = d_data[index + 3 * blockDim.x];
int a4 = d_data[index + 4 * blockDim.x];
int a5 = d_data[index + 5 * blockDim.x];
int a6 = d_data[index + 6 * blockDim.x];
int a7 = d_data[index + 7 * blockDim.x];
d_data[index] = (a + a1 + a2 + a3 + a4 + a5 + a6 + a7);}
```

there is only one thread bundle. Based on this, the thread bundle expansion method with interleaved prefix sum is proposed.

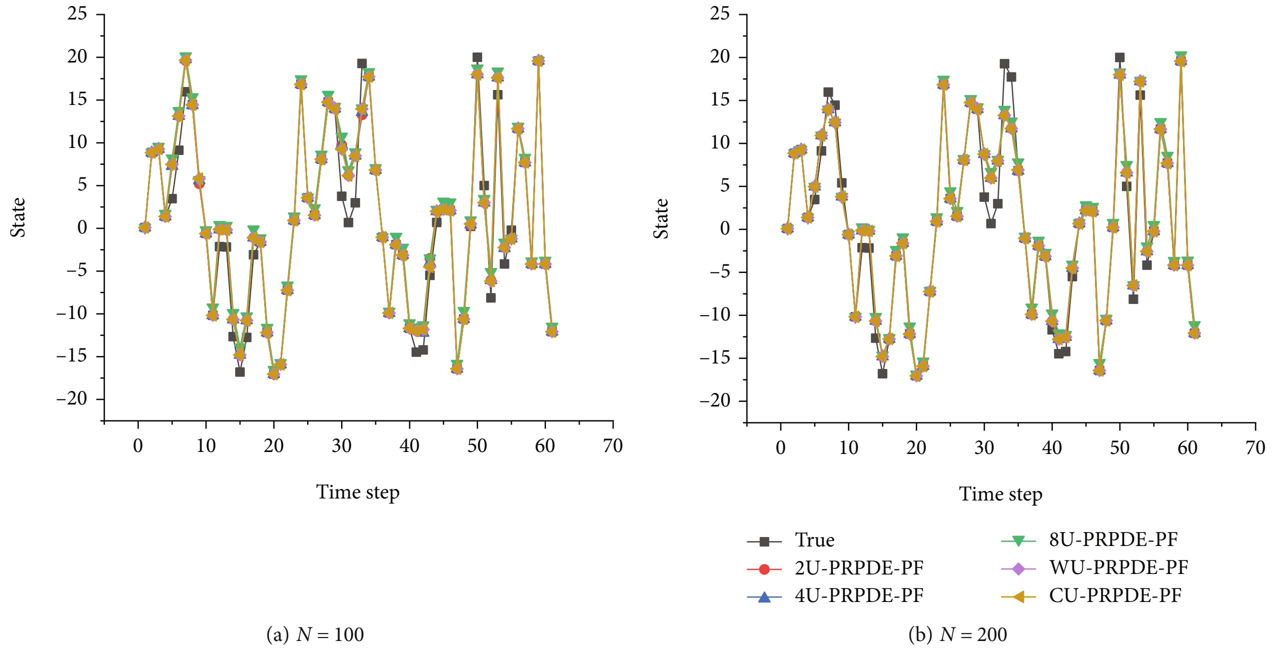
Through the previous experimental analysis, the iterative loop below 32 threads is unfolded. In fact, because of the length limit of the thread block (generally 1024), the number of loops is determined, so the loop can be fully unfolded, i.e., 1024, 512, 256, 128, and 64, and calculated, and the only thing that needs to be noted is that each calculation should be synchronized afterwards. Table 4 shows the pseudocode for a fully expanded loop.

4. Experiment and Performance Analysis

In order to verify the basic performance of the parallel algorithm with the improved prefix sum, the performance of the algorithm is simulated using a typical one-dimensional nonlinear system model and compared with the parallel prefix sum based on the unfolding cycle, parallel prefix sum based on the thread unfolding cycle, and parallel prefix sum based on the full unfolding filtering algorithm. The experimental platform includes the Win10 64-bit system, Visual Studio 2013 programming software, and CUDA9.2-based programming framework, where the GPU

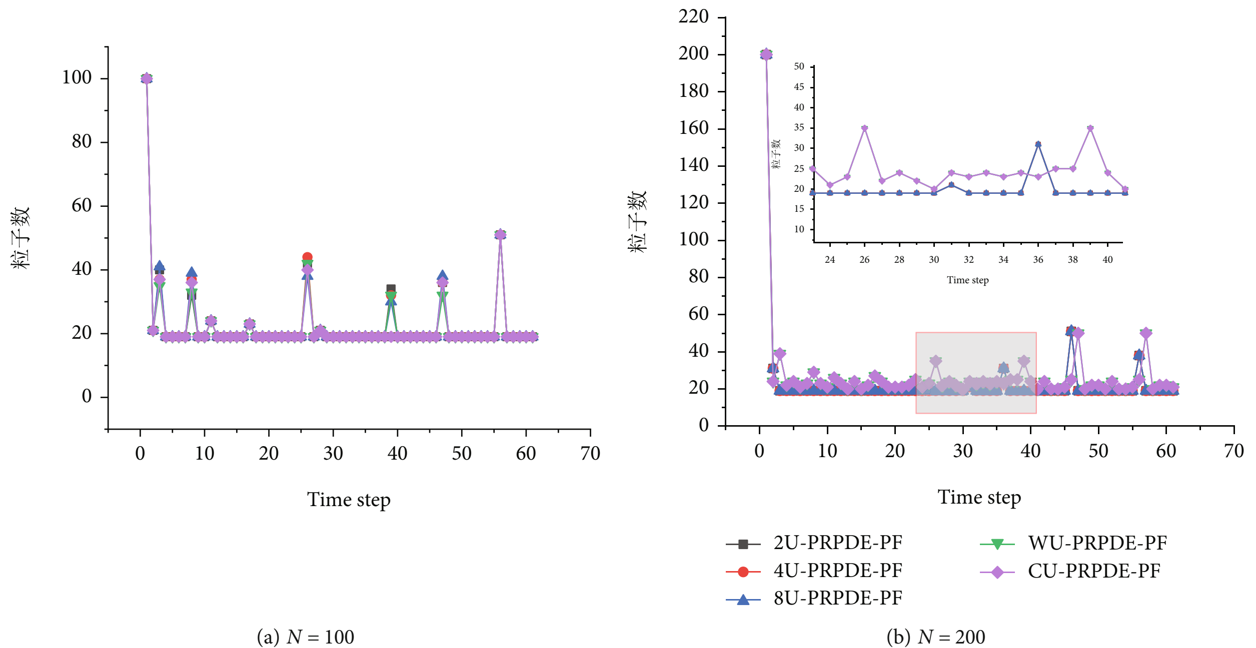
Table 5: The detailed parameters of the experimental platform.

GPU		CPU	
GTX1080Ti		Intel® Core™ i5-4460	
Stream processor unit	3584	CPU	Intel® i5-4460
Video memory	11 GB	Core number	4
Clock frequency	1582 MHz	Memory	8 GB
Memory bit width	352 bits	Clock frequency	3.2 GHz



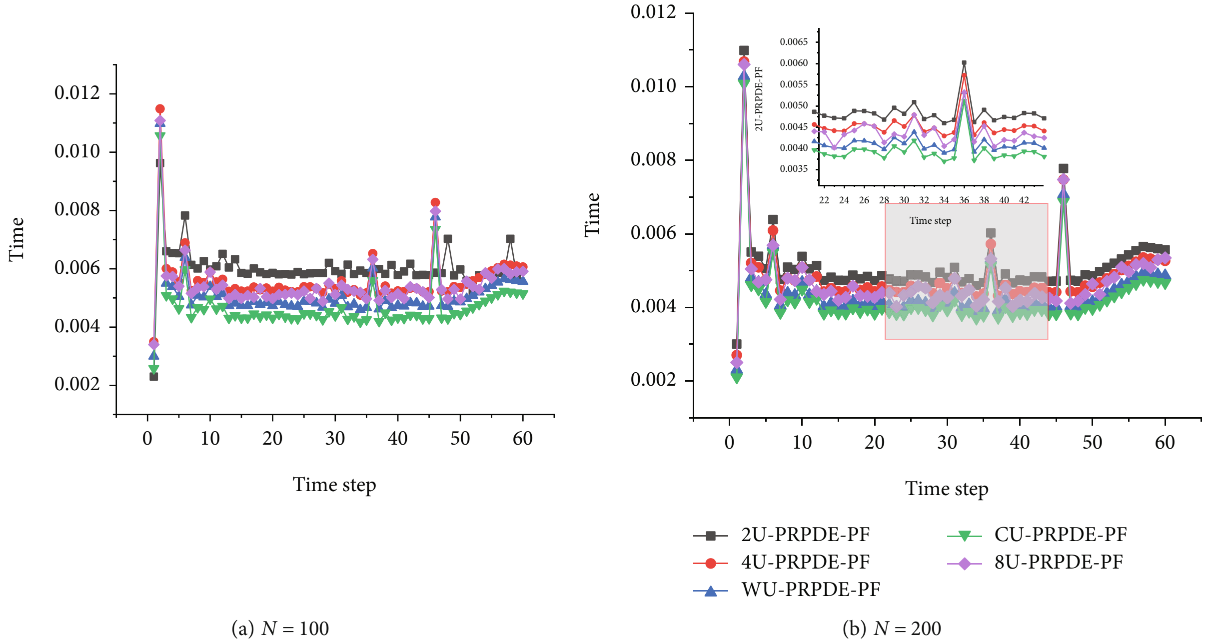
(a) $N = 100$ (b) $N = 200$

Figure 4: State estimation results of five improved algorithms.



(a) $N = 100$ (b) $N = 200$

Figure 5: Particle number curves of the five improved algorithms.



(a) $N = 100$

(b) $N = 200$

FIGURE 6: Time curves calculated by five algorithms.

TABLE 6: Comparison of filter calculation time for $N = 200$.

	2U-PRPDE-PF	4U-PRPDE-PF	8U-PRPDE-PF	WU-PRPDE-PF	CU-PRPDE-PF
Time (s)	0.005012	0.004815	0.00462	0.004465	0.00425

is GTX1080Ti and the CPU is i5-4460. Detailed parameters are listed in Table 5.

The one-dimensional nonlinear system model is as follows:

$$\begin{cases} x_k = 1 + \sin(0.04\pi k) + 0.5x_{k-1} + u_{k-1}, \\ y_k = \begin{cases} 0.2x_k^2 + v_k & (1 \le k \le 30), \\ 0.5x_k\text{-}2 + v_k & (30 < k \le T), \end{cases} \end{cases} \tag{5}$$

System noise of the model is $u_{k-1} \sim \Gamma(3, 2)$, total observation time is$T = 60$, crossover probability is$CR = 0.6$of the evolutionary algorithm, and the maximum evolution time is iteration number$G_{\max} = 10$. In this paper, we use the parallel prefix and expansion factors 2, 4, and 8 (2U-PRPDE-PF, 4U-PRPDE-PF, and 8U-PRPDE-PF) based on the expansion loop. In this paper, the comparison experiments are conducted among the three algorithms, i.e., PF, 8U-PRPDE-PF, warp unrolling PRPDE-PF, and complete unrolling PRPDE-PF.

4.1. Experimental Analysis of Root Mean Square Error. The algorithm is simulated $R_{\mathrm{MC}} = 200$ times by independent Monte Carlo, and the root mean square error of the time is defined as follows:

$$L_k^{\mathrm{RMSE}} = \sqrt{\left(\frac{1}{R_{\mathrm{MC}}}\right) \sum_{j=1}^{R_{\mathrm{MC}}} \left(x_{k,j} - \bar{x}_{k,j}\right)^2}. \tag{6}$$

$x_{k,j}$ and $\bar{x}_{k,j}$ denote the actual and predicted states at the moment k in the jth simulation, respectively. The measurement noise $v_k \sim N(0, 0.001)$. Figure 4 gives a comparison of the momentary root mean square error of the five algorithms for the two settings of the particle number $N = 100$ and $N = 200$.

The performance of the algorithm state estimation is basically the same. It can be seen from Figure 4 that the mean square error of the five improved algorithms, 2U-PRPDE-PF, 4U-PRPDE-PF, 8U-PRPDE-PF, WU-PRPDE-PF, and CU-PRPDE-PF, under the same experimental conditions of the particle number, is reduced relative to the IIPRPDE-PF algorithm, and all of them can guarantee the state estimation ability with the accuracy of the algorithm improved to some extent, indicating that the improved methods improve the state tracking performance of the filtering algorithm to some extent.

4.2. Particle Distribution and Calculation Time Experiments. Figures 5 and 6 show the curves of particle number variation and computation time of the improved prefix sum algorithm based on the unfolding cycle for 60 simulation moments. The comparison of the simulation curves in Figure 5 shows that the particle numbers of 2U-PRPDE-PF, 4U-PRPDE-PF, 8U-PRPDE-PF, WU-PRPDE-PF, and CU-PRPDE-PF decrease gradually and adjust the numbers adaptively with time. In Figure 6, at the time, it can be seen that the time of CU-PRPDE-PF is lower than that of the other filters due to performing full unfolding, fully improving the recursive

Table 7: Computation time of the five parallel algorithms.

N	Computation time (s)				
	CRPF	Block CRPF	2U-PRPDE-PF	4U-PRPDE-PF	Optimized block CRPF
1024	0.15862	0.1125	0.03751 (2.99x)	0.03564 (3.15x)	0.1461
2048	0.21301	0.1676	0.05 (3.35x)	0.048 (3.49x)	0.1747
3200	0.26709	0.21225	0.0625 (3.396x)	0.06 (3.53x)	0.24
4096	0.32074	0.256	0.075 (3.41x)	0.0692 (3.69x)	0.291
6400	0.44983	0.3672	0.096 (3.825x)	0.0783 (4.689x)	0.4076

Table 8: Six algorithms' running schedule. Unit: ms.

N	IIPRPDE-PF	2U-PRPDE-PF	4U-PRPDE-PF	8U-PRPDE-PF	WU-PRPDE-PF	CU-PRPDE-PF
2^{10}	39.2	37.51	36.35	35.64	33.56	33
2^{11}	51.7	50.592	49.011	48	44.2	43
2^{12}	76.05	75.888	73.5165	69.2	65	64
2^{13}	152.35	139.536	135.1755	136.5	130.21	128
2^{14}	250.6	241.536	233.988	225.82	214.18	210
2^{15}	493.3	471.648	456.909	448.2	421.62	414
2^{16}	972	930.24	901.17	883.6	830.77	816
2^{17}	1927.85	1861.296	1803.1305	1748.59	1647.73	1620
2^{18}	3753.85	3607.536	3494.8005	3410.6	3208.42	3154
2^{19}	7526.55	7215.888	6990.3915	6842	6432.94	6324.8
2^{20}	15031	14394.24	13944.42	13662	12847	12631

Table 9: Acceleration ratios of the five algorithms relative to IIPRPDE-PF.

N	2U-PRPDE-PF	4U-PRPDE-PF	8U-PRPDE-PF	WU-PRPDE-PF	CU-PRPDE-PF
2^{10}	1.04505	1.0784	1.09969	1.16806	1.18788
2^{11}	1.0219	1.05487	1.07708	1.16968	1.20233
2^{12}	1.00213	1.03446	1.09899	1.17	1.18828
2^{13}	1.09183	1.12705	1.11612	1.17003	1.19023
2^{14}	1.03753	1.071	1.10973	1.17004	1.19333
2^{15}	1.04591	1.07965	1.10062	1.17001	1.19155
2^{16}	1.04489	1.0786	1.10005	1.17	1.19118
2^{17}	1.03576	1.06917	1.10252	1.17	1.19003
2^{18}	1.04056	1.07412	1.10064	1.17	1.19019
2^{19}	1.04305	1.0767	1.10005	1.17	1.19001
2^{20}	1.02042	1.07792	1.1002	1.17	1.19

loop, increasing the prefix and execution efficiency, i.e., increasing the execution rate of resampling and computation time consumption, while the time of 2U-PRPDE-PF, 4U-PRPDE-PF, 8U-PRPDE-PF, and WU-PRPDE-PF is smaller than that of IIPRPDE-PF with a decreasing trend.

After improving the recursive loop in resampling, the particles are reduced adaptively, and the computation time of the parallel filtering algorithm after all five unfolded loops is relatively reduced and smaller than IIPRPDE-PF. After unfolding the recursive loop within resampling, the overall complexity of the algorithm increases, and the time required for recursive sampling to update the number of particles for calculation in real time is not enough to offset the time saved by the reduction of particles when the number of particles is small, and this situation disappears at the time when the computation time of the parallel differential evolutionary

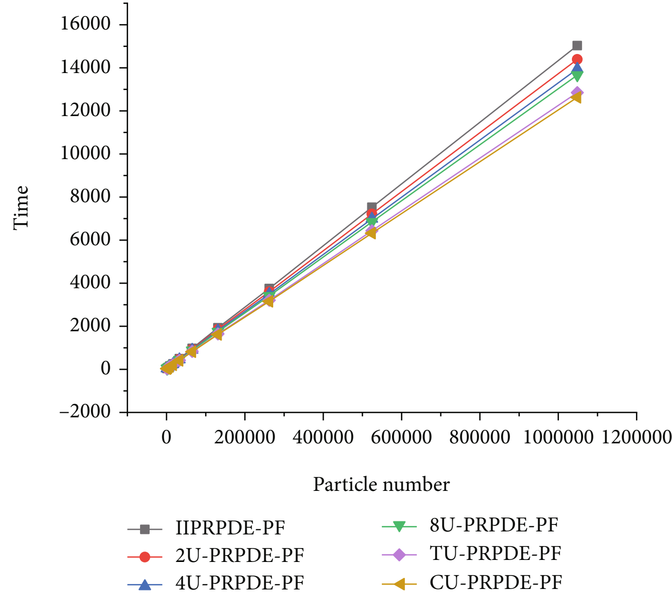


Figure 7: Schedule of the three algorithm runs.

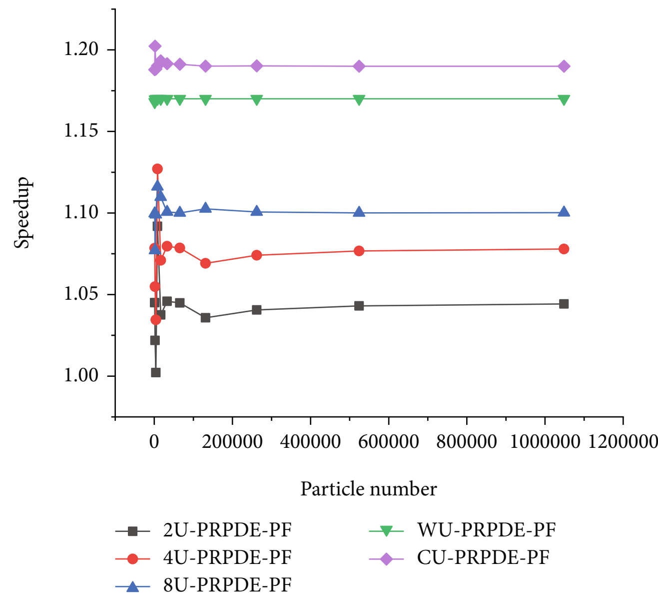


Figure 8: Acceleration ratios of five filtering algorithms with circular unfolding relative to IIPRPDE-PF.

Table 10: The parameters of different GPUs.

GPU	CUDA cores	Base frequency	Video memory	Memory bit width
GTX1080Ti	3584	1.58 GHz	11 GB	352 bits
GTX960	1024	1127 MHz	2 GB	128 bits
GTX950	768	1024 MHz	2 GB	128 bits
GTX750Ti	640	1020 MHz	2 GB	128 bits

particle filter for all unfolding loops is smaller than that of the corresponding parallel differential evolutionary particle filter, which also indicates that the PRPDE-PF sampling of unfolding loops improves the computation time more significantly. The filter computation time shown in Table 6 is obtained by 60 independent Monte Carlo experiments and taking the average of the running time of each filter for each experiment, and it can be seen that CU-PRPDE-PF requires

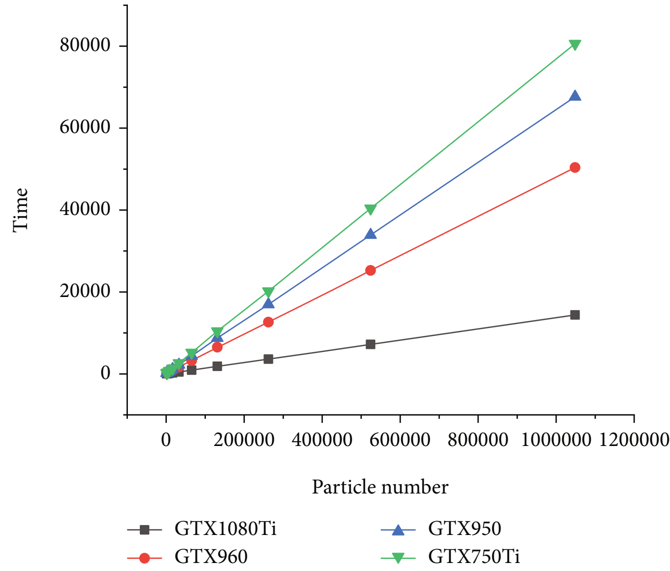


Figure 9: 2U-PRPDE-PF algorithm under different GPUs.

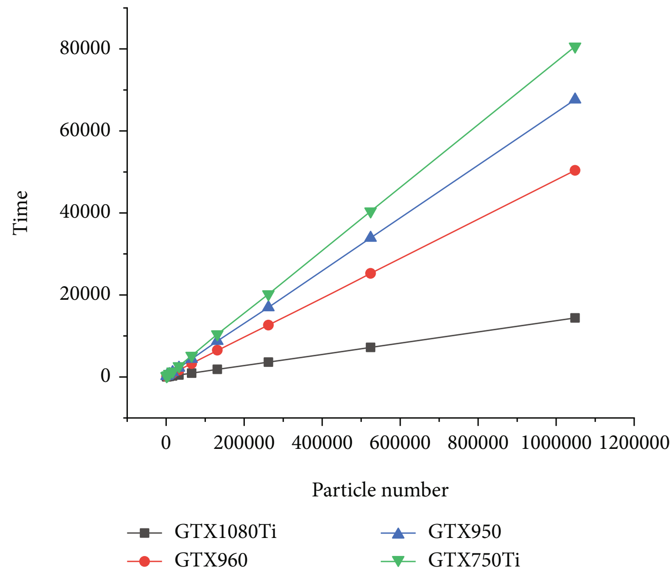


Figure 10: 4U-PRPDE-PF algorithm in different GPUs.

the least computation time. Combining the performance indicators of computational accuracy and computation time of each filter, the fully unfolded loop filter algorithm CU-PRPDE-PF has the least computation time and is the best performance among the five improved filtering algorithms in this paper.

Also, compared with the three smart optimized parallel particle filtering algorithms in the article of Wang et al. [18], the computation time of the improved algorithms (2U-PRPDE-PF, 4U-PRPDE-PF) and the block parallel particle smart optimized particle filtering algorithm in this paper are shown in Table 7, respectively. It can be seen from the table that among the three intelligent optimized parallel algorithms, the block parallel particle filtering algorithm block parallel CRPF has the best performance, followed by the optimized block parallel; the optimization part increases the complexity of the algorithm; and the computational performance decreases compared to the block parallel algorithm. The algorithms proposed in this paper, 2U-PRPDE-PF and 4U-PRPDE-PF algorithms, are compared with the block parallel algorithm, respectively. From Table 7, it is concluded that the improved 2U-PRPDE-PF algorithm in this paper has stronger computational performance than the block parallel CRPF, and a 3.82x acceleration ratio is obtained as the number of particles grows, and the 4U-PRPDE-PF algorithm obtains a speedup ratio of 4.689x as the number of particles increases asymptotically, so the algorithm proposed in this paper has improved

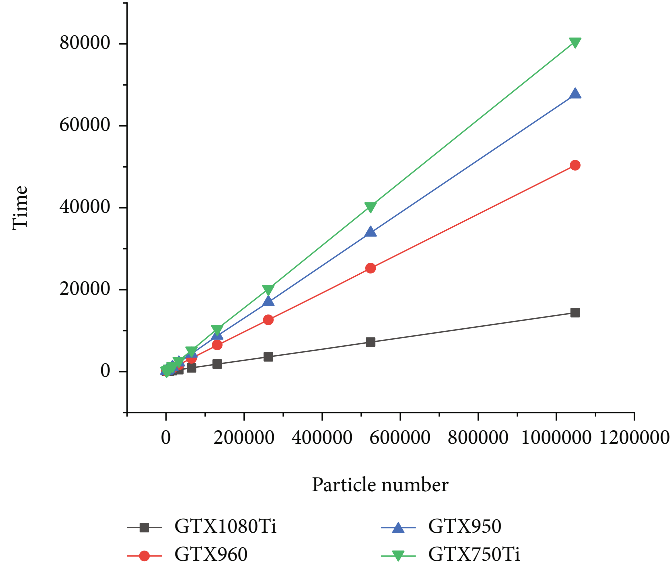


FIGURE 11: 8U-PRPDE-PF algorithm in different GPUs.

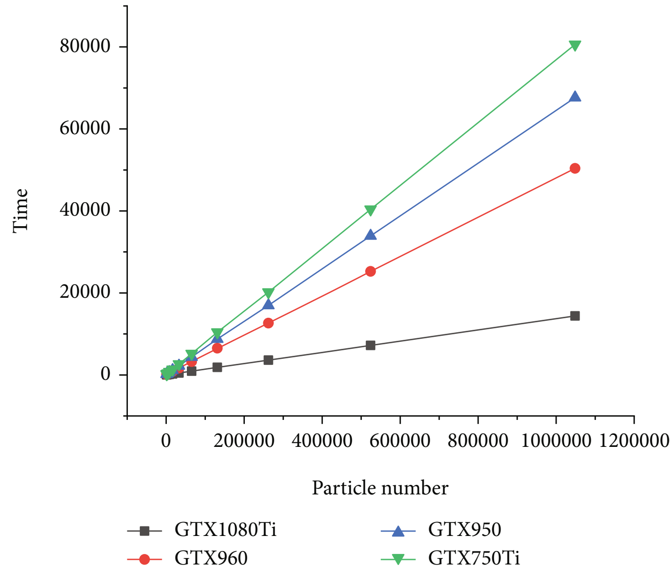


FIGURE 12: TU-PRPDE-PF algorithm in different GPUs.

performance and can obtain a good speedup ratio compared to the block parallel algorithm.

Comparison of the runs of the five parallel differential evolutionary particle filtering algorithms 2U-PRPDE-PF, 4U-PRPDE-PF, 8U-PRPDE-PF, WU-PRPDE-PF, and CU-PRPDE-PF was based on CUDA cyclic unfolding with improved prefixes and postimprovement in the same GPU case. Tables 8 and 9 show the running schedules and speedup ratios of the five improved algorithms, respectively, and Figures 7 and 8 correspond to Tables 8 and 9, respectively, where the speedup ratio is defined as the value obtained by dividing the running time of the original algorithm by the running time of the improved algorithm under the same particle count condition. Figure 8 shows the values obtained by dividing the operation time of the original algorithm IIPRPDE-PF by the operation times of 2U-PRPDE-PF, 4U-PRPDE-PF, 8U-PRPDE-PF, WU-PRPDE-PF, and CU-PRPDE-PF, respectively. The acceleration ratio of CU-PRPDE-PF is the largest and remains around 1.19 as the number of particles increases, while the acceleration ratios of the other four types of 2U-PRPDE-PF, 4U-PRPDE-PF, 8U-PRPDE-PF, and WU-PRPDE-PF eventually remain at a certain value as the number of particles increases. Under GTX1080Ti, the number of particles is 1024; after the direct unfolding with unfolding factors of 2, 4, and 8, from 39.2 ms to 35.64 ms, it can be seen that the direct cyclic unfolding has a very big impact on the efficiency; this is not only because of saving the extra thread block running but also because the

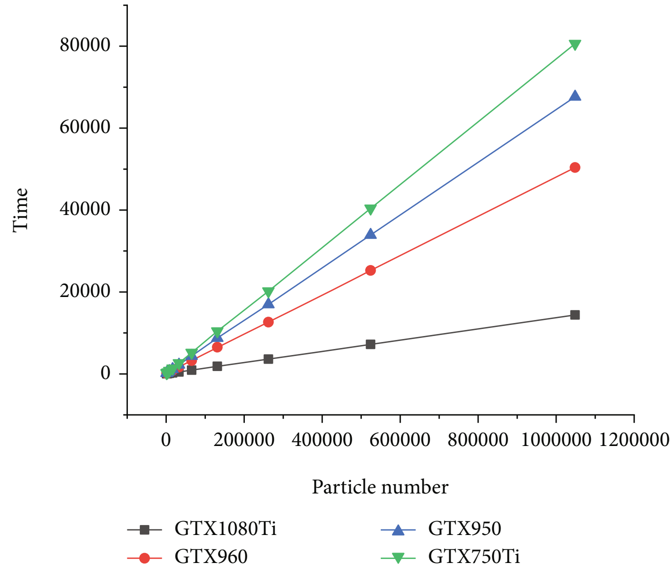


FIGURE 13: CU-PRPDE-PF algorithm in different GPUs.

improvement has more independent memory loading, and storage operations can produce better performance, with better hidden latency. The algorithm in this paper has been improved incrementally to obtain an overall performance improvement of up to 1.19 times.

4.3. Real-Time Performance of Algorithms on Different GPUs. Experimental simulations are performed with the above five improved algorithms based on different GPU conditions. The whole experimental platform includes the Win10 system, Visual Studio 2013 programming software, and CUDA9.2-based programming framework with i5-4460 CPU, listed as Table 10, running the algorithms on four different GPUs with the number of particles from $2^{10}$to 2^{20}.

The performance experiments of the five improved algorithms in this paper are done based on the same CPU and different GPU conditions. According to the analysis in Figures 9–13, compared with the IIPRPDE-PF algorithm, the five improved algorithms of this paper based on IIPRPDE-PF for cyclic unfolding, 2U-PRPDE-PF, 4U-PRPDE-PF, 8U-PRPDE-PF, WU-PRPDE-PF, and CU-PRPDE-PF, exhibit approximately the same growth rate for different GPUs. It is discussed that the acceleration ratio of the algorithm under different GPU conditions is basically proportional to the computational power of the GPU itself, and the performance of the algorithm is optimal under the experimental environment of GPU GTX1080Ti. In this paper, the performance improvement of GPU computation is limited to the improved prefix and problem. Based on the direct segmentation prefix and the improved differential evolutionary particle filtering algorithm, the overall performance improvement speed of CU-PRPDE-PF is up to 19% relative to the IIPRPDE-PF algorithm, and the performance improvement factor of CU-PRPDE-PF can reach up to 1.45 compared with that of the original PDE-PR algorithm. The main reason for the limited performance improvement of the improved algorithm is just not simply parallel on the GPU and requires complex operations or even contains quite a few logical judgments. However, some performance gains can be achieved by prefixing and incremental improvements.

5. Conclusion

In this paper, we propose a CUDA unfolding loop-based state estimation method for differential evolutionary particle filtering to address the problem of inefficient parallel differential evolutionary particle filtering with parallel execution threads and improve the execution efficiency of the prefix sum by unfolding the prefix sum method with an unfolding loop and a thread bundle. The proposed method uses the segmented prefixes after the unfolding loop and the improved resampling and the latest moment of observation to update the proposed distribution of the optimized particle filter in real time and adaptively adjusts the number of particles to be sampled for the particle filter to a smaller number using differential evolutionary resampling. In addition, for the execution of the particle filtering algorithm, the prefix and execution have the problem of inefficient thread execution, and the GPU does not have the branch prediction capability, at every branch it performs, so the algorithm removes the thread bundle differentiation and thread idleness existing in the parallel prefix by unfolding the loop and unfolding the thread bundle method, eliminating the lag caused by the failure of judgment and branch prediction, further improving the overall computational performance. The current CUDA compiler cannot do this optimization for us and requires artificially unfolding the loop within the kernel function, which can greatly improve the kernel performance. The purpose of unfolding the loop in CUDA is twofold: to reduce instruction consumption and to increase the performance by adding more independent scheduling instructions to reduce fragmentation. Simulation results show that the parallel differential evolutionary particle filtering algorithm with this unfolding loop can effectively improve intelligent

optimal particle filtering for nonlinear system states and real-time performance. Finally, experimental simulations show that the algorithm with the improved prefix sum can achieve the best speedup factor of 1.19 relative to the IIPRPPDE-PF algorithm and 1.48 relative to the PDE-PF algorithm under GTX1080Ti, and the experimental data show that the overall performance of the algorithm under different GPUs is proportional to the GPU. The experimental data show that the overall performance of the algorithm under different GPUs is proportional to the GPU computational power, which indicates that the improved algorithm in this paper has universal applicability.

Data Availability

The data used to support the findings of this study are included within the article.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

References

[1] Y. U. Chunchao, Z. Yang, Z. Xia, X. Yuan, and M. Yan, "Heterogeneous source image alignment based on mutual information and particle swarm algorithm using GPU parallel architecture," *Infrared Technology*, vol. 38, no. 11, pp. 938–946, 2016.

[2] H. W. Li, J. Wang, and H. T. Su, "Improved particle filter based on differential evolution," *Electronics Letters*, vol. 47, no. 19, pp. 1078-1079, 2011.

[3] S. W. X. J. C. Jiazhong and Y. Shengsheng, "GPU-based parallel algorithm for particle filtering," *Journal of Huazhong University of Science and Technology (Natural Science Edition)*, vol. 5, pp. 63–66, 2011.

[4] S. K. Das, C. Mazumdar, and K. Banerjee, "GPU accelerated novel particle filtering method," *Computing*, vol. 96, no. 8, pp. 749–773, 2014.

[5] L. M. Murray, A. Lee, and P. E. Jacob, "Parallel resampling in the particle filter," *Journal of Computational & Graphical Statistics*, vol. 25, no. 3, pp. 789–805, 2016.

[6] V. P. Jilkov and J. Wu, "Efficient GPU-accelerated implementation of particle and particle flow filters for target tracking," *Journal of Advances in Information Fusion*, vol. 10, no. 1, pp. 73–88, 2015.

[7] W. Liu, Z. Meng, and D. Xue, "A multi-feature fusion video target tracking algorithm based on CUDA and particle filtering," *Computer System Applications*, vol. 22, no. 11, pp. 123–128, 2013.

[8] S. Lalwani, H. Sharma, S. C. Satapathy, K. Deep, and J. C. Bansal, "A survey on parallel particle swarm optimization algorithms," *Arabian Journal for Science and Engineering*, vol. 44, no. 4, pp. 2899–2923, 2019.

[9] F. Bourennani, "Cooperative asynchronous parallel particle swarm optimization for large dimensional problems," *International Journal of Applied Metaheuristic Computing*, vol. 10, no. 3, pp. 19–38, 2019.

[10] X. Lin and Y. Wu, "Parameters identification of photovoltaic models using niche-based particle swarm optimization in parallel computing architecture," *Energy*, vol. 196, p. 117054, 2020.

[11] S. S. C. B. Fraser, *Computing Included and Excluded Sums Using Parallel Prefix*, Doctoral dissertation, Massachusetts Institute of Technology, 2020.

[12] H. Tokura, T. Fujita, K. Nakano, Y. Ito, and J. L. Bordim, "Almost optimal column-wise prefix-sum computation on the GPU," *The Journal of Supercomputing*, vol. 74, no. 4, pp. 1510–1521, 2018.

[13] G. Thakur, H. Sohal, and S. Jain, "A novel parallel prefix adder for optimized Radix-2 FFT processor," *Multidimensional Systems and Signal Processing*, vol. 3, 2021.

[14] M. Harris and M. Garland, "Optimizing parallel prefix operations for the Fermi architecture," in *GPU Computing Gems Jade Edition*, pp. 29–38, Morgan Kaufmann, 2012.

[15] A. Pîrjan, "Optimization solutions for the segmented sum algorithmic function," *Wseas Us*, 2013.

[16] E. Wynters, "Parallel particle swarm optimization can solve many optimization problems quickly on GPUs," *Journal of Computing Sciences in Colleges*, vol. 33, no. 6, pp. 114–123, 2018.

[17] "Analysis of dimensionality reduction techniques on big data, a novel PCA–whale optimization-based deep neural network model for classification of tomato plant diseases using GPU".

[18] J. Wang, J. Cao, W. Li, P. Yu, and K. Huang, "A novel parallel accelerated CRPF algorithm," *Applied Intelligence*, vol. 50, no. 3, pp. 849–859, 2020.