Received 31 December 2022, accepted 15 January 2023, date of publication 23 January 2023, date of current version 8 February 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3238873

RESEARCH ARTICLE

Accelerated Particle Filter With GPU for Real-Time Ballistic Target Tracking

DAEYEON KIM[1], YUNHO HAN[2], HEONCHEOL LEE[3], (Member, IEEE), YUNYOUNG KIM[4], HYUCK-HOON KWON[4], CHAN KIM[4], AND WONSEOK CHOI[4]

[1]School of Electronic Engineering, Kumoh National Institute of Technology, Gumi-si 39177, South Korea
[2]Department of IT Convergence Engineering, Kumoh National Institute of Technology, Gumi-si 39177, South Korea
[3]Department of IT Convergence Engineering, School of Electronic Engineering, Kumoh National Institute of Technology, Gumi-si 39177, South Korea
[4]PGM Research and Development Laboratory, LIG Nex1, Seongnam-si 13488, South Korea

Corresponding author: Heoncheol Lee (hclee@kumoh.ac.kr)

This work was supported by the Theater Defense Research Center Funded by Defense Acquisition Program Administration under Grant UD200043CD.

ABSTRACT This study addresses the problem of real-time tracking of high-speed ballistic targets. Particle filters can be used to overcome the nonlinearity of motion and measurement models in ballistic targets. However, applying particle filters (PFs) to real-time systems is challenging since they generally require a significant computation time. So, most of the existing methods of accelerating PF using a graphics processing unit (GPU) for target tracking applications have accelerated computation weight function and resampling part. However, the computational time per part varies from application to application, and in this work, we confirm that it takes a lot of computational time in the model propagation part and propose accelerated PF by parallelizing the corresponding logic. The real-time performance of the proposed method was tested and analyzed using an embedded system. And compared to conventional PF on the central processing unit (CPU), the proposed method shows that the proposed method significantly reduces computational time by at least 10 times, improving real-time performance.

INDEX TERMS Ballistic target tracking, graphics processing unit, particle filter, real-time systems.

I. INTRODUCTION

The performance of ballistic target interception highly depends on accurate target tracking. For high-accuracy tracking under measurement uncertainties, state estimation must be adopted based on various filtering algorithms. Generally, measurement model noise is assumed as a Gaussian distribution for mathematical simplicity. However, owing to the nonlinear and non-Gaussian characteristics of the measurement noise caused by the seeker random and scintillation, the assumption of a Gaussian distribution is invalid [1], [2]. In nonlinear and non-Gaussian uncertainties, conventional filtering algorithms may perform unsatisfactorily. For this reason, linear Kalman filter-based target tracking filters may not converge properly or even diverge during interception.

Accordingly, various nonlinear filters have been previously applied to target state estimation, including the extended Kalman filter (EKF), particle filter (PF), and unscented Kalman filter (UKF). Compared to the EKF, PF performs more consistently under nonlinear and non-Gaussian noise [3], [4], [5]. This is because the PF has an inherent capability and flexibility to deal with various types of error distributions. However, the primary difficulty of a PF in a real-time system is the heavy computational burden, as the required number of particles exponentially increases with the number of state variables. The computational issue is a crucial constraint and must be solved for real-time application.

The PF takes more time because the number of iterations increases as the number of particles increases due to the nature of the sampling-based algorithm. However, this repetition is necessary to find an appropriate value using particle information and the particle resampling process, the algorithm progress time can be reduced if the parts that require many calculations are parallelized [6]. Therefore, algorithms such as PFs, which require a significant time to find target information, can be accelerated by parallelization using a

The associate editor coordinating the review of this manuscript and approving it for publication was Utku Kose.

TABLE 1. Related works to particle filter for target tracking.

Related works	GPU-based Parallelization	Parallelization Part	Missile Application
[11],[12]	X	-	X
[4] ,[13]	X	-	○
[14],[17],[18],[20]	○	Weight computation	X
[19],[20]	○	Resampling	X
[15]	○	rendering / normalized	X
[16],[17],[19]	○	Likelihood calculation	X
Ours	○	Model propagation	○

graphics processing unit (GPU) [7]. PFs are accelerated using GPU in studies that require quick results, such as target tracking using sensors or fields used in real-time.

To achieve high-speed target tracking, the PFs are accelerated in Compute Unified Device Architecture (CUDA) with a GPU. If the entire process of the PF algorithm is converted to CUDA, all parts regardless of their computation time are converted. Then to use a PF in a GPU, the data can be handed over to a PF running on the CPU to the GPU required. But this is inefficient. It can lead to a considerable overhead time. Therefore, the identification of parts of the PF requiring a considerable amount of computational time is accelerated using a GPU. Calculation parts that require a considerable amount of computational time are parallelized using a GPU. Thus, the computational time can be further reduced even if overhead is generated.

As the PF algorithm is suitable for tracking, it is widely used. From Table 1, it is used for various tracking tasks such as target tracking [11] and [12], object tracking, and motion tracking. In [4] and [13], they used a PF for missile applications. Acceleration studies were conducted using a GPU to perform the PF algorithm in real-time. In [14] and [18], the PF that is parallelized for the weight computation is proposed. A GPU was used to improve the PF estimation for target tracking rather than acceleration [14], And in [18], a GPU was used to accelerate IoT applications. The tracking algorithm was accelerated by approximately 55 % compared to the CPU-based algorithm. In [16], they proposed a PF that parallelized the likelihood function calculation and reduced the calculation time of that. However, it required time to generate random values, such studies have conducted parallelization in environments with a considerable change in the signal or amount of information of particles, such as image tracking. In [17], [19], and [20], more than one part requiring a long computation time in the research environment or those that could be parallelized independently were parallelized. Examples include weight computation, likelihood function evaluation for calculating the particle state, and resampling. When two or more parts of a PF are parallelized in the GPU, more overhead is generated. Therefore, tasks reducing the overhead time, such as kernels, can be shared. So, unlike previous related works, we propose a method that parallelized part of the measurement acquisition, especially the model propagation part, using GPU.

This paper explains a high-speed target tracking system and the necessity for the PF algorithm acceleration. Parallelization is used in parts of the PF requiring a considerable amount of calculation time to accelerate. This achieves high-speed for target tracking. Methods using a GPU to accelerate parts of the PF and reasons for accelerating those parts are described. The results and analysis of the target-tracking algorithm with the accelerated PF to that of the original target-tracking algorithm with un-accelerated PF are compared.

The contributions of this paper are as follows:

- This is the first approach to accelerate a PF for ballistic target tracking under glint noise.
- To the best of our knowledge, this is the first study to address and analyze the problem of long computation time for the model propagation of the sampling process.
- A new parallelization method was developed for real-time PFs for ballistic target tracking.
- The computation time of the PF was significantly reduced even with the overhead time for the CUDA initialization on a widely-used embedded system.

The remainder of this paper is organized as follows. Section II describes the target missile tracking system based on PFs and the real-time problems of PFs. In Section III, after the computation times for the PF are profiled block-wise, a new parallelization method for the model propagation of the sampling process is proposed. In Section IV, the evaluation results of the proposed method are presented and quantitatively compared with those of other methods using a widely used embedded system. Finally, conclusions are presented in section V.

II. PROBLEM DESCRIPTION

The objective of the target-tracking filter is the real-time estimation of the true target states. To evaluate the performance of the tracking filter and accelerate the system, the target trajectories of ballistic missiles are generated. We consider a target-tracking filter for the reentry phase of a ballistic missile. In the reentry phase, atmospheric drag is a significant force determining the path of the missile. Accordingly, the forces acting on the target during reentry arise from gravity and aerodynamic drag. A ballistic missile is represented as a point mass in three-dimensional Cartesian coordinates. Since we only consider target tracking for the reentry phase, the thrust force is set to zero and the mass of the tracking target is constant. Aerodynamic drag D is expressed as a function of the air density ρ, target velocity V, aerodynamic coefficient $\mathrm{C_D}$, and reference area S.

$$D = \frac{1}{2}\rho V^2 \cdot C_D \cdot S \tag{1}$$

A. MOTION AND MEASUREMENT MODEL FOR PARTICLE FILTER

Since target-tracking estimations are based on the target motion model, several target models have been proposed. In this study, the well-known Singer model was used [8] and [9]. The Singer model assumes that the target acceleration is a zero-mean, first-order, stationary Markov process. The state-space representation of the continuous-time Singer model is:

$$\dot{x} = F\boldsymbol{x} + G\boldsymbol{w} \tag{2}$$

where

$$F = \begin{bmatrix} 0_3 & I_3 & 0_3 \\ 0_3 & 0_3 & I_3 \\ 0_3 & 0_3 & -I_3/\tau \end{bmatrix} \; and \; G = \begin{bmatrix} 0_3 \\ 0_3 \\ I_3 \end{bmatrix} \tag{3}$$

Here, $\boldsymbol{x}$ is the state of target, $\boldsymbol{w}$ is the zero-mean white Gaussian noise. And τ and I_3 in $\boldsymbol{F}$ represent the maneuver time constant and identity matrix of order 3. Its discrete-time equivalent is as follows:

$$\boldsymbol{x}_k = \Phi_{k-1}\mathbf{x}_{k-1} + \boldsymbol{w}_{k-1}, \text{ with } \boldsymbol{w}_{k-1} \sim N\left(0, Q_k\right) \tag{4}$$

$$\Phi_k \simeq I + F\Delta t \tag{5}$$

$$Q_k \simeq S_w Q_0 = S_w \begin{bmatrix} \frac{\Delta t^5}{20} I_3 & \frac{\Delta t^4}{8} I_3 & \frac{\Delta t^3}{6} I_3 \\ \frac{\Delta t^4}{8} I_3 & \frac{\Delta t^3}{3} I_3 & \frac{\Delta t^2}{2} I_3 \\ \frac{\Delta t^3}{6} I_3 & \frac{\Delta t^2}{2} I_3 & \Delta t I_3 \end{bmatrix} \tag{6}$$

where Φ_k and Δt represents state transition matrix and sampling time interval. Covariance Q_k in Eq. 6 consists of the power spectral density S_w and white noise jerk model Q_0. The acceleration increment over a time period is the integral of the jerk over the period.

For the state-space representation in Eq. 2, $\boldsymbol{x}$ denotes the associated variable of the target position, velocity, and acceleration. $\boldsymbol{P}$, $\boldsymbol{V}$, and $\boldsymbol{A}$ in Eq. 7 are the target position, velocity, and acceleration, respectively, in Cartesian coordinates.

$$\boldsymbol{x} = \left[\boldsymbol{P}^T \; \boldsymbol{V}^T \; \boldsymbol{A}^T\right]^T \tag{7}$$

$$\boldsymbol{P} = \left[x \; y \; z\right]^T \tag{8}$$

where (x, y, z) represent the target position in the Cartesian coordinate system. Three measurements achieved by the radar were assumed: elevation, azimuth, and range. The measurements were acquired with respect to the target and radar position. In Eq. 9, the subscript r denotes the relative position between the target and radar. m represents the position of the radar. Consequently, two bearing angles z_θ, z_ψ and the relative range z_R can be represented as nonlinear equations as Eq. 10 using states in Cartesian and radar noises.

$$\left[x_r \; y_r \; z_r\right]^T = \left[x \; y \; z\right]^T - \left[x_m \; y_m \; z_m\right]^T \tag{9}$$

$$\begin{bmatrix} z_\theta \\ z_\psi \\ z_R \end{bmatrix} = \begin{bmatrix} \tan^{-1}\left(z_r \Big/ \sqrt{x_r^2 + y_r^2}\right) + n_{G,\theta} + n_\theta \\ \tan^{-1}\left(y_r / x_r\right) + n_{G,\psi} + n_\psi \\ \sqrt{x_r^2 + y_r^2 + z_r^2} + n_R \end{bmatrix} \tag{10}$$

where n_θ, n_ψ, and n_R represents the receiver noise of the radar, and $n_{G,\theta}$ and $n_{G,\psi}$ are non-Gaussian glint noises generated in radar measurements [3].

B. THE PROBLEM OF ALGORITHM ACCELERATION

For high-speed targets such as ballistic missiles, the filter update rate and estimation accuracy are crucial. Because precision guidance and control lead to a successful interception, accurate target tracking is an indispensable element. In this study, a PF was used for higher estimation accuracy and consistency. However, the heavy computational burden of PF should be solved for real-time application. To cope with the problem, we propose a GPU-based acceleration method for PFs.

The iterations of the parts in the PF algorithm were processed as many times as the number of particles. And the entire PF algorithm was iterated a predetermined number of times by the user. If the PF is iterated 300 times, the model propagation and weighting function are calculated by iterating as many times as the number of particles for each iteration.

If the PF algorithm proceeds using a CPU, the calculations are sequentially performed as the number of iterations. As the number of particles increases, the computation time increases accordingly. In the CPU, the repetitive calculation in the PF algorithm was carried out as many times as the number of particles. Whereas in the GPU, the same calculation could be parallelized and calculated simultaneously. Therefore, when using a GPU, the parts requiring a considerable amount of calculation time in the iterations can be significantly reduced. If the appropriate parallelization technique is applied, the larger the number of particles, the shorter the calculation time compared to that of the CPU.

III. PROPOSED METHOD

A. OVERVIEW OF THE PROPOSED METHOD

The PF flowchart for target tracking is shown in Fig. 1. Initially, the particles were sprinkled at random intervals within the measurement range. The model propagation step predicts how the target will move. The movement is estimated using Eq. 11. In Eq. 11, x_p is the acceleration, speed, and location information of the initial particles; *randnum* is a matrix of random values; Q_k is a 9×9 matrix of the filter covariance and can be represented by Eq. 13. In Eq. 13, sig_{acc} is the signal accuracy of the filter covariance; dt is the time the state of the target change. and the matrix A is the target state transition matrix, which is defined in Eq. 5. x_{mid1} was obtained using these values. The matrix was used to estimate the state of the target. This process is performed for each particle. All the obtained information in x_{bar} is added and used in the filter update part. After the model propagation step, glint and sensor noise models generated values.

$$x_{bar} = (A \times \boldsymbol{x}_p) + (A \times \sqrt{Q_k} \times randnum) \tag{11}$$

$$Q_k = 2 \times {sig_{acc}}^2/tau \times \begin{bmatrix} dt^5/20 \times eye\,(3) & dt^4/8 \times eye\,(3) & dt^3/6 \times eye\,(3) \\ dt^4/8 \times eye\,(3) & dt^3/3 \times eye\,(3) & dt^2/2 \times eye\,(3) \\ dt^3/6 \times eye\,(3) & dt^2/2 \times eye\,(3) & dt \times eye\,(3) \end{bmatrix} \tag{12}$$

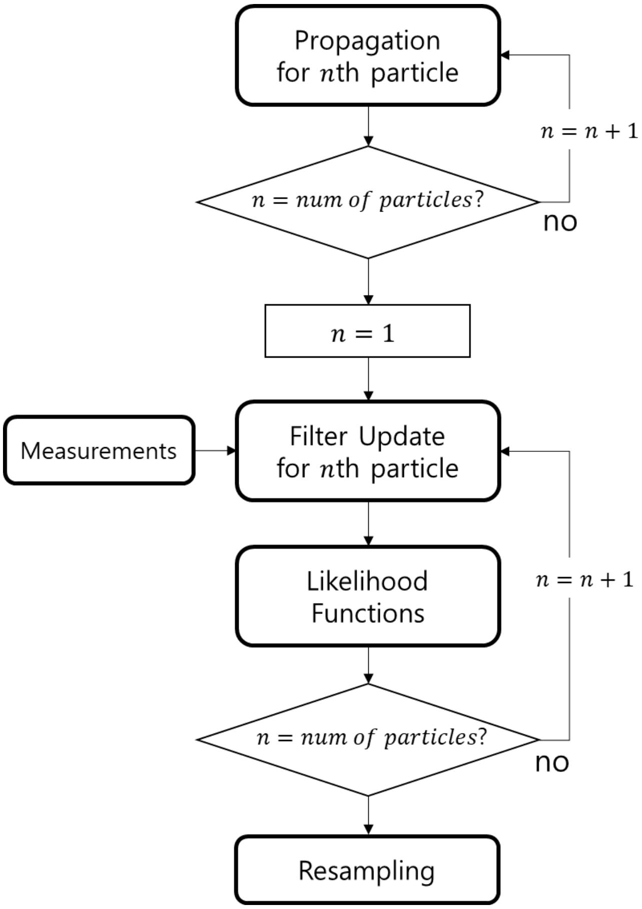

Propagation for nth particle
$n = n + 1$
$n = num\ of\ particles?$
no
$n = 1$
Measurements
Filter Update for nth particle
Likelihood Functions
$n = n + 1$
$n = num\ of\ particles?$
no
Resampling

FIGURE 1. Flowchart describing the PF algorithm for target missile tracking using only CPU.

B. COMPUTATION TIME PROFILING

To accelerate the PF, the part of the PF algorithm requiring a considerable amount of calculation time should be identified. Computation time measurement for each part was performed in Nvidia Jetson Xavier, and the results are summarized in Table 2, showing that the model propagation part takes more time than the other parts. The computation time for each part of PF shown in Table 2 is the result of an experiment with 5000 particles.

The calculation proceeded in the model propagation part consists of obtaining the square root of the filter covariance value, adding, and multiplying matrixes. The matrix A and Q_k are matrixes of size 9 × 9. The $\boldsymbol{x}_p$ is a matrix with 9 rows and columns as large as the number of particles. The model propagation part is calculated for one column of the $\boldsymbol{x}_p$ matrix at one iteration. As shown in Fig. 1, the matrix calculation process in the model propagation part is repeated as many as the number of particles. Since the matrix calculation process, which is iterated as many as the number of particles, is performed every single iteration of the particle filter algorithm, the model propagation part is taken the most computation time. The filter update part and likelihood function part in Fig. 1 are also iterated as many as the number of particles, but since the two parts are simple numerical operations rather than matrix operations, so the computation time of the two parts is not taken much time compared to the model propagation part.

TABLE 2. The Computation time for each part of PF in only the CPU.

Part	Computation time(s)
Create a true target model and measurement	0.102
Generate particles	0.102
Model propagation	24.841
Create noise	0.002
Calculate weight function	0.972
Resampling	1.902
Confidence	0.127

C. PARALLELIZED PARTICLE FILTER 1.0

The PF was accelerated by performing parallel calculations using Eq. 11, which progresses in the model-propagation part. As described in Part B, when the number of particles used in the PF algorithm is large, the calculated matrices become large. Therefore, a significant amount of calculation time is required for the model propagation part. However, Eq. 11 does not consist of complex equations compared to time-consuming equations. Thus, it is easy to parallelize using a GPU and can be accelerated effectively.

For CUDA, tasks such as CUDA initialization and malloc are executed first. The matrices to be used for input are copied to the variables defined by CUDA to be used. The GPU memory is required as shown in Fig. 2, according to the size of the matrices calculated. The matrix has 9 rows and 1024 threads in one block, and the more particles are used, the more GPU memory uses. As shown in Fig. 2, the memory usage of the GPU must be defined to store values in the memory designated when using the CUDA kernel and calculate them in parallel.

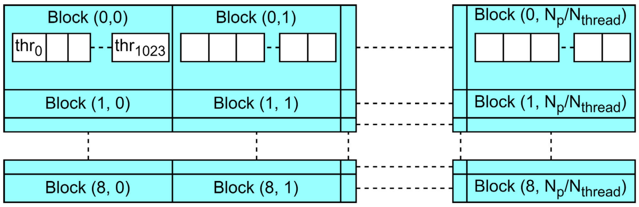


FIGURE 2. **GPU memory allocation for CUDA kernel use.**

Fig. 3 shows a flowchart when performing the model propagation part using CUDA. The kernels are calculated in parallel using GPU. In the first kernel, as shown in Fig. 2, the random values according to the normal distribution are generated using the "curand_normal" function provided by the CUDA library. To put different seeds for each thread, the time when the kernel is performed and the thread ID that is generated by using ParticleID in Fig. 2 are used as seeds and applied to the "curand". The random value, which is the result of kernel execution, is generated as a matrix in which rows are 9 and columns are as many as the number of particles. Because the size of the random value matrix should be adjusted for matrix operation to be applied in Kernel 4. Kernel 1 is also used in the parallelized PF 2.0 described in section III-D.

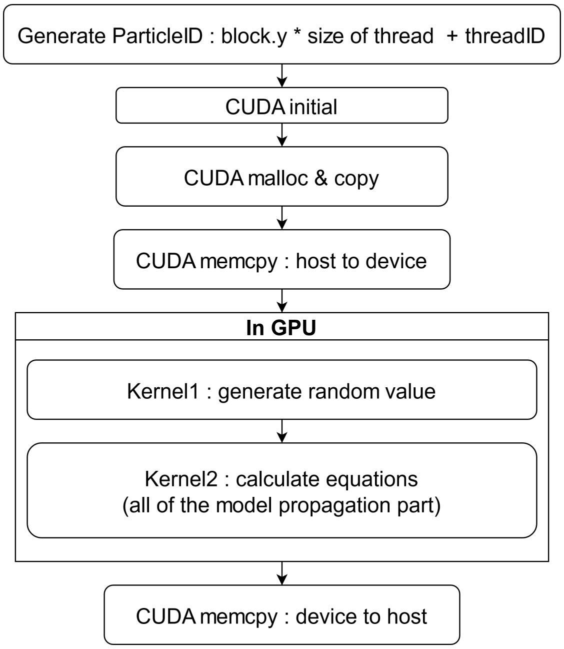

Generate ParticleID : block.y * size of thread + threadID
CUDA initial
CUDA malloc & copy
CUDA memcpy : host to device
In GPU
Kernel1 : generate random value
Kernel2 : calculate equations
(all of the model propagation part)
CUDA memcpy : device to host

FIGURE 3. **Flowchart of parallelized model propagation using integrated kernels in PF algorithm.**

To deal with the matrixes using Kernel 2, it is necessary to generate IDs that specify the addresses of the values of the matrixes. Fig. 5 is shown how to create IDs, which are defined equally within all kernels used in this part and the D part.

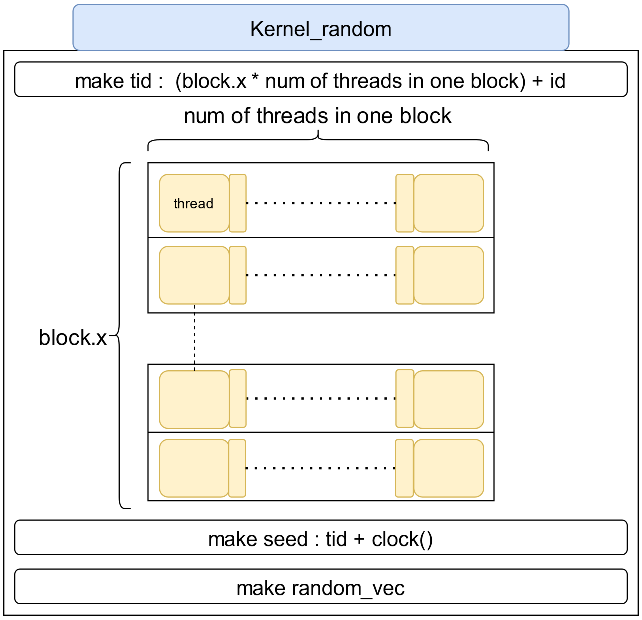

Kernel_random
make tid : (block.x * num of threads in one block) + id
num of threads in one block
thread
block.x
make seed : tid + clock()
make random_vec

FIGURE 4. **Random values generated in CUDA Kernel 1 are shown in Fig. 3. We define it as the number of particles and store the generated random value in each tid.**

The target matrixes of the kernels have a size of 9 × 9 or a size in which rows are 9 and columns are as many as the number of particles. It is the same as the ParticleID in Fig. 3,and its size is the same as the number of particles. Therefore, ID can assign as many addresses as the number of particles in the column. StateID assigns the address of a matrix row that has 9 rows and StateIDy assigns the address of a matrix column. The generated IDs in Fig. 5 are used to assign addresses to elements of the matrixes to be targeted by kernels so that the values of each address can be calculated in parallel. In this part, the parallelized PF 1.0 which is a method of using a kernel integrated with all the functions used in Eq. 11 is described. Kernel 2, all equations in Eq. 11 are calculated, which is described in detail in Fig. 6. The addresses of the matrixes calculated in Kernel 2 are assigned as the size of the target matrixes of calculations in Eq. 11. Since the CreateIDs in Fig. 5 are declared in the first order in Kernel 2, addresses for the values of the matrixes can be assigned using stateID, StateIDy, and ID according to the size of the matrixes used in each calculation. It is difficult to calculate Eq. 11 as one equation as in CPU because the size of the result matrixes of each calculation is different. This is also the reason for defining the IDs with different sizes and addresses in Fig. 5. The variables $\boldsymbol{x}_{mid1}$, $\boldsymbol{x}_{mid2}$, and $\boldsymbol{x}_{mid3}$ defined within Kernel 2 in Fig. 6 are necessary for storing the result matrixes having a different size for this reason. After performing Kernel 2, the $\boldsymbol{x}_{mp}$ matrix can be obtained as a result of the parallelized model propagation.

D. PARALLELIZED PARTICLE FILTER 2.0

Eq. 11 consists of a square root operation of a matrix, multiplication, and addition between matrixes. The method

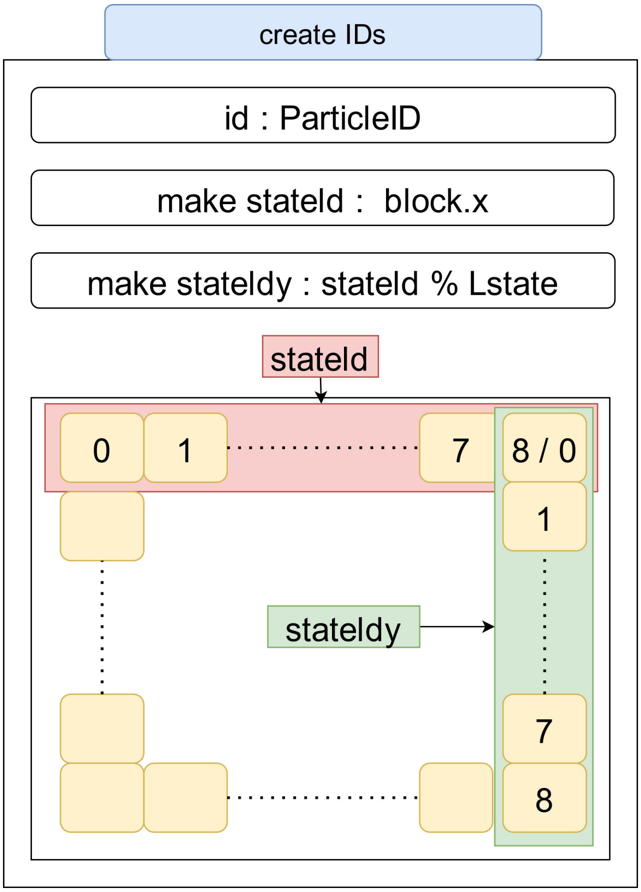


FIGURE 5. Create IDs used to calculate matrixes. Each id is defined to obtain the values' addresses of the matrixes.

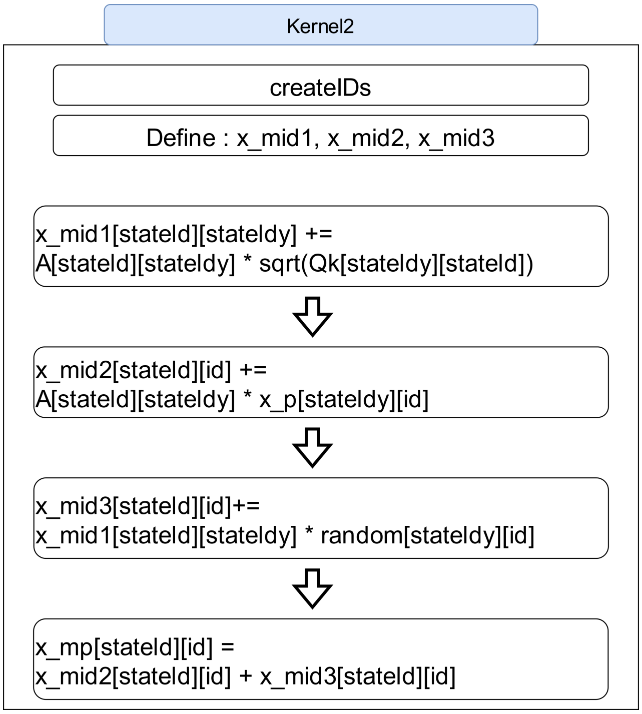

Kernel2
createIDs
Define : x_mid1, x_mid2, x_mid3
x_mid1[stateId][stateIdy] +=
A[stateId][stateIdy] * sqrt(Qk[stateIdy][stateId])
x_mid2[stateId][id] +=
A[stateId][stateIdy] * x_p[stateIdy][id]
x_mid3[stateId][id]+=
x_mid1[stateId][stateIdy] * random[stateIdy][id]
x_mp[stateId][id] =
x_mid2[stateId][id] + x_mid3[stateId][id]

FIGURE 6. The Kernel is defined for using CUDA to compute Eq. 11.

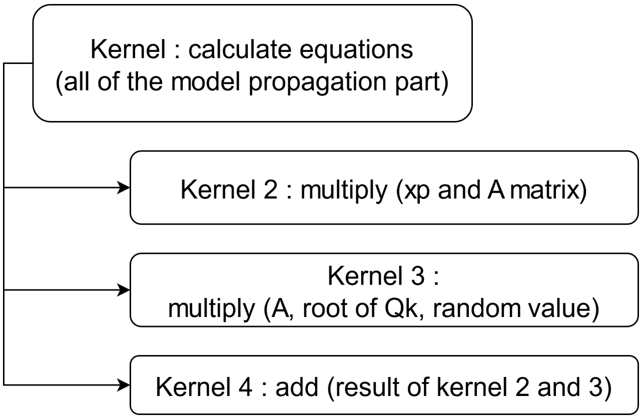

Kernel : calculate equations
(all of the model propagation part)
Kernel 2 : multiply (xp and A matrix)
Kernel 3 :
multiply (A, root of Qk, random value)
Kernel 4 : add (result of kernel 2 and 3)

FIGURE 7. Presents Kernel 2 in Parallelized PF 1.0 as three kernels.

described in Part C performs parallel calculations by kernelizing Eq. 11 by integrating it into one kernel. As shown in Fig. 2, this method was used by defining two kernels. Thus, the variables to store the results of each equation are defined in the kernel. Therefore, each time the particle filter is iterated, the task of defining variables within the kernel for storing the result matrixes is iterated. This means that the task of allocating the space in GPU to store the values of the variables is performed every iteration.

To reduce the time required for these tasks, a method of securing the space in GPU by declaring variables in advance before using the kernel was devised. Declared variables should be used as inputs or outputs of the CUDA kernels. Since the calculations constituting Eq. 11 are not complicated, Kernel 2 in Fig. 5 was subdivided, and parallel calculations were performed. Subdivided kernels are defined so that predefined variables can be used as inputs and outputs of the kernels described in this part. The particle filter of this method is proposed 2.0, and Fig. 7 reconstructs the kernel of Fig. 6 and proposes three kernels. The proposed PF 2.0 is performed by subdividing Kernel 2 in the proposed PF 1.0 into three kernels.

The subdivided kernels are shown in Fig. 8, 9, and 10. In Kernel 2 in this part, as shown in Fig. 8, the matrix multiplication of A and $\boldsymbol{x}_p$ is calculated in parallel. Since the size of A matrix is 9 × 9, the address of the row is assigned using stated and the address of the column is assigned using stateIDy. The multiplication of matrixes is added after the values of the rows in the preceding matrix and the columns in the following matrix are multiplied. Therefore, the row address of $\boldsymbol{x}_p$, which is the following matrix, is assigned as stateIDy to calculate the multiplication of matrixes. And the size of the column in the $\boldsymbol{x}_p$ is the number of particles, so the addresses of the columns are assigned using ID. And then, as shown in Fig. 9, three matrixes are multiplied in Kernel 3. First, multiply A matrix by the square root of Q matrix and store it in $\boldsymbol{x}_{mid1}$matrix. The size of the two matrixes is 9 × 9, so the size of $\boldsymbol{x}_{mid1}$ matrix is the same. And the *random* matrix is multiplied by a random value matrix generated by Kernel 1 and stored in $\boldsymbol{x}_{mid3}$. In Kernel 4, result matrixes of Kernel 2 and Kernel 3 are added in parallel, as shown in Fig. 10. Since both resultant matrixes have a size that rows of 9 and columns are as many as the number of particles, the result matrix, $\boldsymbol{x}_{bar}$, is obtained as a matrix of the same size.

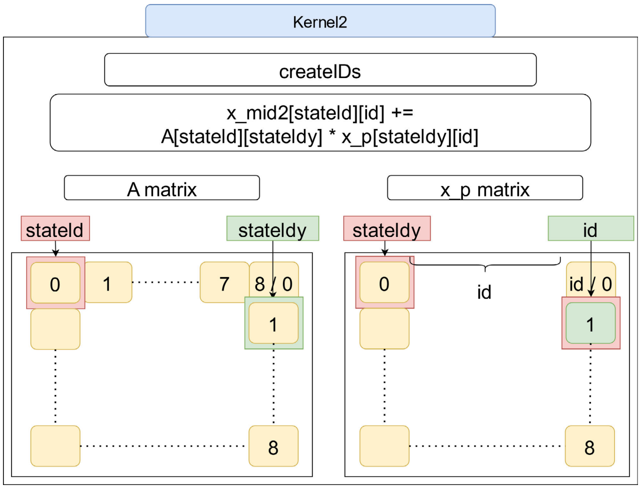


FIGURE 8. Multiplication A matrix and x_p matrix in Eq. 11.

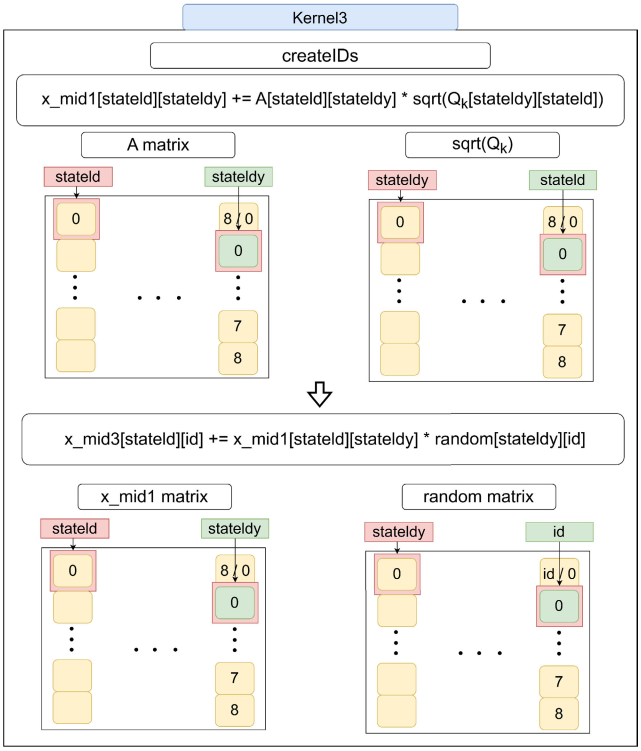

Kernel3
createIDs
x_mid1[stateId][stateIdy] += A[stateId][stateIdy] * sqrt(Q_k[stateIdy][stateId])
A matrix
sqrt(Q_k)
stateId
stateIdy
0
8
0
7
8
stateIdy
stateId
0
8
0
7
8
x_mid3[stateId][id] += x_mid1[stateId][stateIdy] * random[stateIdy][id]
x_mid1 matrix
random matrix
stateId
stateIdy
0
8
0
7
8
stateIdy
id
0
id
0
7
8

FIGURE 9. Multiplication A matrix, square root of Q_{temp} matrix and x_p matrix in Eq. 11. After calculating the multiplication of the two matrixes, the resulting matrix and random value matrix are multiplied.

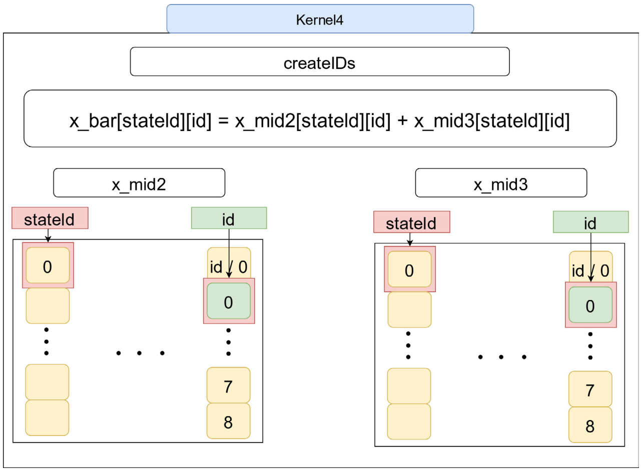

Kernel4
createIDs
x_bar[stateId][id] = x_mid2[stateId][id] + x_mid3[stateId][id]
x_mid2
x_mid3
stateId
id
0
id
0
7
8
stateId
id
0
id
0
7
8

FIGURE 10. Addition of result matrixes from Kernel 2 and Kernel 3. The x_{bar} matrix, the result of Kernel 4, is the result matrix of the model propagation part.

Algorithm 1 Parallelized Particle Filter 2.0

```
procedure $\boldsymbol{x}_{estimated} = \mathrm{PPF}(\boldsymbol{x}_p)$
• Generate particles
  for $i = 1$: simulation time do
    • Model propagation:
    do in parallel:for $N = 1$: num of particles
      $\boldsymbol{x}^N_{mid2} = A \times \boldsymbol{x}_p$                //kernel 2
    do in parallel:for$N = 1$: num of particles
      $\boldsymbol{x}^N_{mid1} = A \times \sqrt{Q}_k$              //kernel3
      $\boldsymbol{x}^N_{mid3} = \boldsymbol{x}^N_{mid} \times randnum$
    do in parallel:for $N = 1$: num of particles
        $\boldsymbol{x}^N_{bar} = \boldsymbol{x}^N_{mid2} + \boldsymbol{x}^N_{mid3}$          //kernel 4
    • Create sensor noise
    • Create glint noise
    • Measurement acquisition
    • Calculate weight function
    for $j = 1$: Particles do
      • Filter update
      • Associated likelihood functions
    end for
    • Resampling
  end for
end procedure
```

Anyway, since the sequential particle filter method proceeds to calculate Eq. 11 and verifies whether it has been repeated as many as the target number of particles, it can be said that the time complexity is determined by the number of particles. An algorithm with linear time complexity can reduce calculation time by eliminating overlapping calculations, so a parallelized particle filter will show excellent performance. In addition, from the perspective of space complexity, the optimal memory required for the algorithm is allocated in advance according to Fig. 2. However, in the proposed PF 1.0, we reallocate space to store the resulting matrix in the process of all operations in Eq. 11 at once, which results in poor space complexity. Therefore, the proposed PF 2.0 pre-allocates a place to store computational results, which makes space complexity better, and efficiently uses GPU resources.

IV. RESULTS

In this study, GPU-based accelerated PF for high-speed target tracking was performed by parallelizing the model propagation process in PF. First, the simulation results of ballistic target tracking can be described by glint noise. Furthermore, the proposed PF algorithm performs faster calculations compared to the results of the CPU based on the acceleration of various GPUs. Compared to the other methods of PF to parallelize resampling or likelihood functions, the only way to show good performance in our applications was ours.

A. RESULTS OF HIGH-SPEED TARGET TRACKING

The effectiveness of the proposed acceleration method is assessed in a ballistic target tracking scenario. For the numerical simulation, the dynamic model in Eq. 1 and Eq. 2 were used to set the true reference trajectory. The aerodynamic drag and weight of the missile were set as in [10]. The sampling interval was set to $\Delta t = 0.01$s, with 200 intervals, yielding a total simulation time of 2s. The standard deviations of the radar receiver noise models n_θ, n_ψ, and n_R were 0.1°, 0.1°, and 1 m, respectively. Glint noise $n_{G,\theta}$ and $n_{G,\psi}$ are mixtures of Gaussians, which follow the distribution.

$$p = (1-\varepsilon)\, p_{G_1} + \varepsilon p_{G_2} \tag{13}$$

where ε is the glint probability. p_{G_1} and p_{G_2} are Gaussians in $p_{G_1} \sim N(0, 0.5^2)^\circ$ and $p_{G_2} \sim N(0, 1^2)^\circ$ at the range of 100m in respectively [3]. The tracking motion model follows the Singer model in Eq. 5 and the measurement model is expressed by Eq. 13. The position of the radar is assumed to be fixed on the ground. Whereas the ballistic target moves at a high speed, considering the gravity and aerodynamic drag. As a result, the velocity of the ballistic target varies with the simulation time.

The resulting trajectory and the estimated results are represented in Fig. 11–13. The number of particles is 15000, which shows a satisfactory tracking performance.

B. RESULTS OF ALGORITHM ACCELERATION WITH GPU

We constructed the following experiments on the embedded system, Jetson Xavier NX: computational time measurements of each algorithm for the number of particles. The parallelized algorithms have significantly faster performance, And the larger the number of particles used in the particle filter algorithm, the less time the particle filter calculated in parallel using GPU takes to computation. The performance of the PF algorithm for the entire algorithm is shown in Table 3. Here, for parallelized computation using CUDA, overhead is inevitable, the overhead includes CUDA initialization, variables definition for using CUDA, kernels definition, etc. However, even if the overhead time is included, it takes more time when the algorithm is performed using only the CPU.

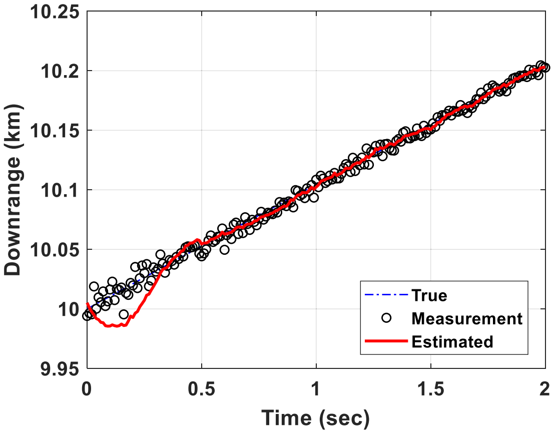


FIGURE 11. Estimated target downrange compared to downrange calculated by radar measurements and true position.

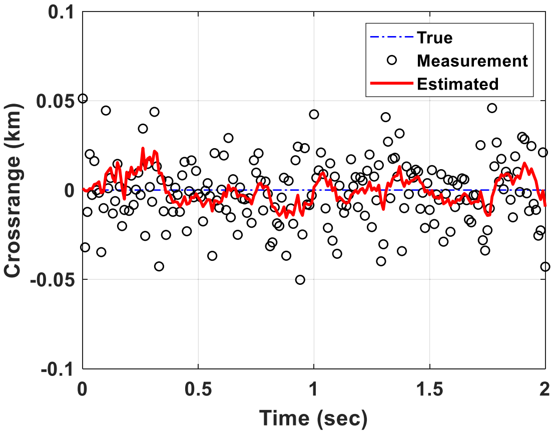


FIGURE 12. Estimated target cross range compared to cross range calculated by radar measurements and true position.

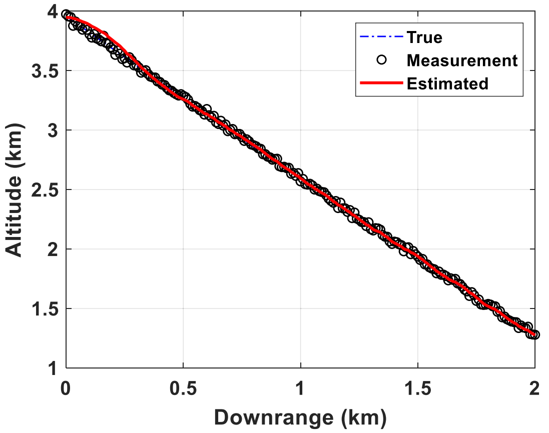


FIGURE 13. Estimated target altitude compared to altitude calculated by radar measurements and true position.

Fig. 15 shows a comparison of the calculation time of the entire algorithm using only CPU and using both CPU and GPU with CUDA when the number of particles is 5000. The PF algorithm takes almost the same amount of time to perform the entire algorithm, and the other parts, except the PF, have a short time. This result shows that the calculation time

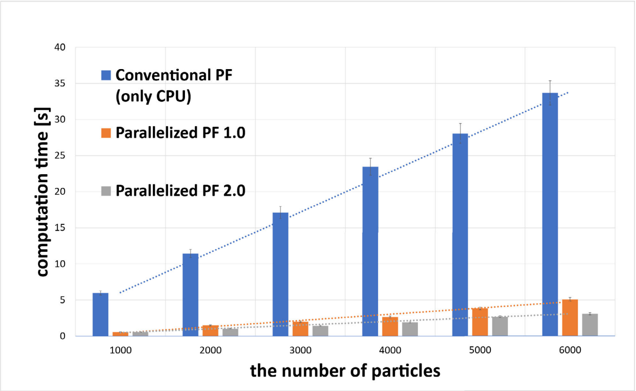


FIGURE 14. Performance time of the entire algorithm in a Xavier environment. The parallelized PF algorithm using GPU kernels required much less performance time.

TABLE 3. Comparison of PF computation times.

Number of Particles	Only CPU (s)	Proposed 1.0 (s)	Proposed 2.0 (s)	Overhead(s)	
				1.0	2.0
1000	5.598	0.575	0.568	0.100	0.098
2000	11.438	1.485	1.035	0.100	0.105
3000	17.117	1.991	1.422	0.101	0.099
4000	23.468	2.651	1.920	0.098	0.105
5000	28.048	3.843	2.669	0.109	0.117
6000	33.680	5.091	3.111	0.104	0.114

decreases significantly when parallelization using CUDA is used in the model propagation part, where it takes the most time. And Fig. 15 shows the speedup between the conventional PF and the parallelized PF algorithm 2.0. The difference in the performance time between the conventional PF and the proposed PF 2.0 is the largest in the model propagation part.

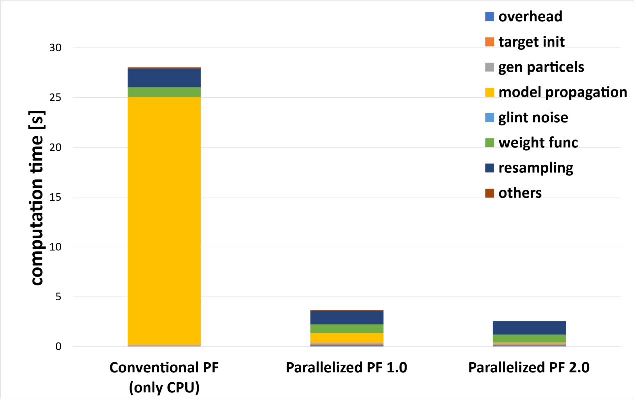


FIGURE 15. Time of the entire algorithm in a Xavier environment. When the target tracking algorithm was applied, the method performed only the CPU required the longest calculation time in the model propagation part. The calculation time using the GPU was dramatically reduced in this part.

Other methods include parallelization of resampling or parallelization of likelihood functions, which are compared and shown in Fig. 16. In conclusion, other methods have not only very slow computational time for model propagation, but also show that the performance may be lower than that of conventional PF due to overhead, and our method achieves optimal performance by selectively parallelizing the parts that need acceleration through computational time profiling in advance. Most importantly in this result, comparing different parallelization methods by applying them to our applications may result in unfair results. Other methods are designed for applications that are different from ours, and the input of the application may be different, finally, they will eventually show optimal performance in their applications. After all, what this shows is that parallelize for computation should be applied differently for each application. In other words, each acceleration algorithm can ensure optimal results when configured with the appropriate acceleration algorithm through profiling of its applications. So, we conducted profiling about conventional PF, and we found that the model propagation part takes most of the whole computation time. And the parallelization of model propagation has shown better performance in our applications.

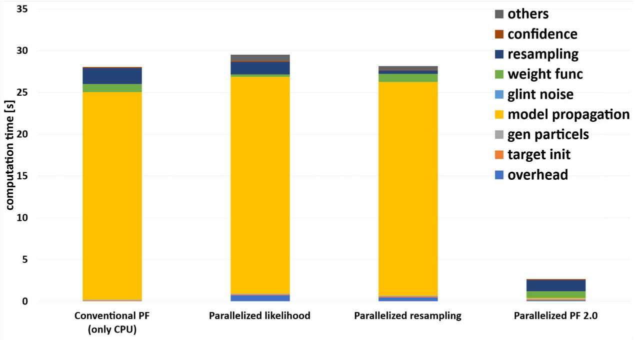


FIGURE 16. Time of the algorithms including other PF methods in a Xavier environment. The calculation time in the proposed PF 2.0 was dramatically reduced in this part.

Table 4 is an extension of the above experiments, and is the result of profiling the performance of a particle filter with 5000 particles using other embedded boards, Jetson AGX Orin, and several GPU cards. The hardware specification was higher, the better performance was shown. For example, the computation time with Jetson AGX Orin was smaller than the computation time with Jetson Xavier NX because Jetson AGX Orin has higher hardware specification than Jetson Xavier NX. Instead, the local memory usage with Jetson AGX Orin was larger than the local memory usage with Jetson Xavier NX, which indicates that there exists a trade-off relation between the computation time and the local memory usage. In addition, the computing peak performance values on the Geforce RTX 3070 and 3090 were 2.57 % and 2.54 %, respectively, resulting in bottlenecks due to the robust performance of the GPU compared to the performance required by the proposed method. Therefore, one should select carefully

TABLE 4. Comparison results of evaluation in various GPUs.

Model	SM Count	Memory Size (GB)	Computation Time(s)	Peak-Performance (%)		Local Memory Usage (MB)	Achieved Occupancy (%)	Speedup (x)
				Computing	Memory			
Intel i7-10700 CPU	-	-	28.048	-	-	-	-	1.000
NVIDIA Jetson Xavier NX	**6**	**7.59**	**2.669**	**57.75**	**63.52**	**6.750**	**89.52**	**10.513**
NVIDIA Jetson AGX Orin	16	29.82	1.559	27.36	28.33	13.500	63.33	17.991
NVIDIA Geforce RTX 3070	46	7.79	1.063	2.57	19.10	38.812	62.51	26.386
NVIDIA Geforce RTX 3090	82	23.68	1.066	2.54	5.38	69.187	68.17	26.311

a hardware for algorithm acceleration by considering the hardware cost and the trade-off relation. The most important point in Table 4 is that the conventional particle filter only on CPU can be highly speeded up by the proposed parallelized particle filter regardless of the selection of the GPU-based hardware system.

C. DISCUSSION

The computation time of the target tracking algorithm was compared when using only the CPU and when using parallelized particle filter 1.0 and 2.0. the result is that the entire algorithm to which the parallelized particle filter was applied had much less computation time than when the algorithm was performed using only the CPU. And when comparing two methods using GPU, it took less time to compute when using the proposed PF 2.0 method.

The reasons for the result of comparing the proposed PF 1.0 and 2.0 are as follows. First, since most of the overhead time occurs in CUDA initialization, if the number of kernels or variables used is not significantly different, the overhead time is similar. For the difference to occur in overhead time, used kernels that perform more complex computations are integrated, or inputs and outputs are defined a lot compared to the experimental environment of this paper. however, the difference in the number of used kernels and variables in the two methods described in this paper is not significant. Therefore, the occurrence time of the overhead is similar. The parallelized particle filter 2.0 uses predefined variables in which the values of the result matrix will be stored to move the result matrixes of the kernels. By defining variables to be used in advance, the area where the values will be stored has been set. However, the parallelized particle filter 1.0 method defines variables every iteration when the kernel is executed to sets the area where the variables will store. Therefore, the second reason is a difference in that variables are defined in advance, or the variables are defined every iteration. For this reason, the parallelized particle filter 1.0 method takes more computation time in the model propagation part than the proposed PF 2.0 method.

V. CONCLUSION

In this study, the first approach was developed to accelerate a PF for target missile tracking. A PF algorithm was used to track the high-speed moving ballistic target, and acceleration was performed using a GPU to achieve real-time performance. For the ballistic target, a PF was used to track the state of the target, such as its movement and angle, and it was successfully estimated without significant differences.

Most of the time was spent in the PF algorithm in the target tracking algorithm, especially in the model propagation part of the information held by the particles. The part identified as requiring a lot of computation time was parallelized by CUDA using a GPU. The result of parallelization was that the computation time was reduced compared to the algorithm using only a CPU, even considering the overhead time inevitably occurring when using CUDA. The algorithms with the parallelized PF proposed in this study using a GPU require less computation time than estimating the state of the ballistic target using only the CPU. Both methods using GPU can be said to have much more real-time performance than when the entire algorithm is performed using only the CPU. Comparing the two methods using GPU, using the proposed PF 2.0 method is more effective because the calculation is not complicated and the variables to be used in GPU are predefined.

REFERENCES

[1] G. M. Siouris, *Missile Guidance and Control Systems*. Berlin, Germany: Springer, 2004, pp. 113–119.

[2] G. Hewer, R. Martin, and J. Zeh, "Robust preprocessing for Kalman filtering of glint noise," *IEEE Trans. Aerosp. Electron. Syst.*, vol. AES-23, no. 1, pp. 120–128, Jan. 1987.

[3] J. Kim, M. Tandale, P. K. Menon, and E. Ohlmeyer, "Particle filter for ballistic target tracking with glint noise," *J. Guid., Control, Dyn.*, vol. 33, no. 6, pp. 1918–1921, Nov. 2010.

[4] E. J. Ohlmeyer and P. K. Menon, "Applications of the particle filter for multi-object tracking and classification," in *Proc. Amer. Control Conf.*, Jun. 2013, pp. 6181–6186.

[5] W. Youn and H. Myung, "Robust interacting multiple model with modeling uncertainties for maneuvering target tracking," *IEEE Access*, vol. 7, pp. 65427–65443, 2019.

[6] G. Hendeby, R. Karlsson, and F. Gustafsson, "Particle filtering: The need for speed," *EURASIP J. Adv. Signal Process.*, vol. 2010, no. 1, pp. 1–9, Dec. 2010.

[7] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. Lefohn, and T. Purcell, "A survey of general-purpose computation on graphics hardware," *Comput. Graph. Forum*, vol. 26, no. 1, pp. 80–113, Mar. 2007.

[8] R. Singer, "Estimating optimal tracking filter performance for manned maneuvering targets," *IEEE Trans. Aerosp. Electron. Syst.*, vol. AES-6, no. 4, pp. 473–483, Jul. 1970.

[9] X. R. Li and V. P. Jilkov, "Survey of maneuvering target tracking. Part I. Dynamic models," *IEEE Trans. Aerosp. Electron. Syst.*, vol. 39, no. 4, pp. 1333–1364, Oct. 2003.

[10] D. C. Wright and T. Kadyshev, "An analysis of the north Korean nodong missile," *Sci. Global Secur.*, vol. 4, no. 2, pp. 129–160, Feb. 1994.

[11] M. R. Morelande and S. Challa, "Manoeuvring target tracking in clutter using particle filters," *IEEE Trans. Aerosp. Electron. Syst.*, vol. 41, no. 1, pp. 252–270, Jan. 2005.

[12] P. M. Djuric, M. Vemula, and M. F. Bugallo, "Target tracking by particle filtering in binary sensor networks," *IEEE Trans. Signal Process.*, vol. 56, no. 6, pp. 2229–2238, Jun. 2008.

[13] M. Yu, L. Gong, H. Oh, W.-H. Chen, and J. Chambers, "Multiple model ballistic missile tracking with state-dependent transitions and Gaussian particle filtering," *IEEE Trans. Aerosp. Electron. Syst.*, vol. 54, no. 3, pp. 1066–1081, Jun. 2018.

[14] R. Cabido, D. Concha, J. J. Pantrigo, and A. S. Montemayor, "High speed articulated object tracking using GPUs: A particle filter approach," in *Proc. 10th Int. Symp. Pervasive Syst., Algorithms, Netw.*, 2009, pp. 757–762.

[15] E. Murphy-Chutorian and M. M. Trivedi, "Particle filtering with rendered models: A two pass approach to multi-object 3D tracking with the GPU," in *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit. Workshops*, Jun. 2008, pp. 1–8.

[16] B. Goyal, T. Budhraja, R. Bhatnagar, and C. Shivakumar, "Implementation of particle filters for single target tracking using CUDA," in *Proc. 5th Int. Conf. Adv. Comput. Commun. (ICACC)*, Sep. 2015, pp. 28–32.

[17] X. Tang, J. Su, F. Zhao, J. Zhou, and P. Wei, "Particle filter track-before-detect implementation on GPU," *EURASIP J. Wireless Commun. Netw.*, vol. 2013, no. 1, pp. 1–9, Dec. 2013.

[18] S. Kim, J. Cho, and D. Park, "Moving-target position estimation using GPU-based particle filter for IoT sensing applications," *Appl. Sci.*, vol. 7, no. 11, p. 1152, 2017.

[19] R. Cabido, A. S. Montemayor, and J. J. Pantrigo, "High performance memetic algorithm particle filter for multiple object tracking on modern GPUs," *Soft Comput.*, vol. 16, no. 2, pp. 217–230, Feb. 2012.

[20] G. Szwoch, "Performance evaluation of the parallel object tracking algorithm employing the particle filter," in *Proc. Signal Process., Algorithms, Archit., Arrangements, Appl.*, 2016, pp. 119–124.

[21] J. Wu and V. P. Jilkov, "Parallel multitarget tracking particle filters using graphics processing unit," in *Proc. 44th Southeastern Symp. Syst. Theory (SSST)*, Mar. 2012, pp. 151–155.

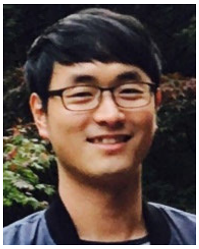
[22] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing," *Proc. IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

[23] M. Harris. *Optimizing Parallel Reduction in CUDA*. [Online]. Available: https://developer.nvidia.com/page/home.html

[24] NVIDIA. *NVIDIA CUDA C Programming Best Practices Guide 2013*. [Online]. Available: http://developer.nvidia.com/

DAEYEON KIM is currently pursuing the B.S. degree in electronic engineering with the Kumoh National Institute of Technology, Gumi, Gyeongsangbuk-do, South Korea. His research interests include deep learning, real-time embedded systems, and algorithm acceleration with GPU and FPGA.

YUNHO HAN received the B.S. degree in electronic engineering from the Kumoh National Institute of Technology, Gumi, Gyeongsangbuk-do, South Korea, in 2021. He is currently pursuing the M.S. degree with Department of IT Convergence Engineering, Kumoh National Institute of Technology. His research interests include path planning, robot navigation, real-time embedded systems, and algorithm acceleration with GPU and FPGA.



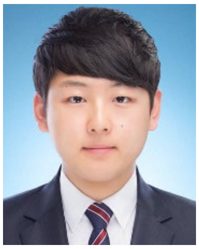
HEONCHEOL LEE (Member, IEEE) received the B.S. degree in electronic engineering and computer sciences from Kyungpook National University, Daegu, South Korea, in 2006, and the M.S. and Ph.D. degrees in electrical engineering and computer sciences from Seoul National University, Seoul, South Korea, in 2008 and 2013, respectively. From 2013 to 2019, he was a Senior Researcher at the Agency for Defense Development, Daejeon, South Korea. Since 2019, he has been an Assistant Professor with the School of Electronic Engineering, Kumoh National Institute of Technology, Gumi, South Korea. He is currently a Technical Adviser with the Robot Navigation Division, Cleaning Science Research Institute, LG Electronics. His research interests include SLAM, robot navigation, machine learning, real-time embedded systems, prognostics, and health management.

YUNYOUNG KIM received the B.S. degree in mechanical engineering from Hanyang University, Seoul, South Korea, in 2014, and the M.S. degree in aerospace engineering from the Korea Advanced Institute of Science and Technology (KAIST), Daejeon, South Korea, in 2016.

Since 2016, she has been a Senior Researcher with LIG Nex1, Seongnam, South Korea. Her research interests include the control and guidance of unmanned aerial vehicle and missile systems, optimal control, and convex optimization.

HYUCK-HOON KWON received the B.S., M.S., and Ph.D. degrees in aerospace engineering from the Korea Advanced Institute of Science and Technology (KAIST), Deajeon, South Korea, in 2002, 2005, and 2020 respectively.

In 2009, he joined the LIG Nex1 for development of precision guided missiles. His research interests include convex optimization, optimal control, guidance and autopilot design, and nonlinear control.



CHAN KIM received the master's degree in information and control engineering from Kwangwoon University, Seoul, in 2015.

From 2015 to 2019, he worked as a researcher in a company related to vehicle parts. Since 2020, he has been working as a Senior Researcher with PGM Research and Development Group Development, LIG Nex1, Gyeonggi-do. His research interests include embedded systems, embedded hw/fw, real-time embedded systems, missile systems, and optimization and acceleration.

WONSEOK CHOI received the M.D. degree in defense convergence engineering from Yonsei University, Seoul, South Korea, in 2017.

From 2015 to 2017, he was a Senior Researcher at LIG Nex1 for PGM Research and Development Group Development, Gyeonggi-do, South Korea. His research interests include embedded systems, embedded SW, real-time embedded systems, and missile systems.