https://doi.org/10.20965/jaciii.2016.p0743

Paper:

Evaluation of an OpenCL-Based FPGA Platform for Particle Filter

Shunsuke Tatsumi*, Masanori Hariyama*, and Norikazu Ikoma**

*Graduate School of Information Sciences, Tohoku University
6-6-05 Aramaki Aza Aoba, Aoba, Sendai 980-8579, Japan
E-mail: {s_tatsumi, hariyama}@ecei.tohoku.ac.jp
**Faculty of Engineering, Nippon Institute of Technology
4-1 Gakuendai, Miyashiro-machi, Minamisaitama-gun, Saitama 345-8501, Japan
E-mail: ikoma@nit.ac.jp
[Received February 20, 2016; accepted June 13, 2016]

Particle filter is one promising method to estimate the internal states in dynamical systems, and can be used for various applications such as visual tracking and mobile-robot localization. The major drawback of particle filter is its large computational amount, which causes long computational-time and large power-consumption. In order to solve this problem, this paper proposes an Field-Programmable Gate Array (FPGA) platform for particle filter. The platform is designed using the OpenCL-based design tool that allows users to develop using a high-level programming language based on C and to change designs easily for various applications. The implementation results demonstrate the proposed FPGA implementation is 106 times faster than the CPU one, and the power-delay product of the FPGA implementation is 1.1% of the CPU one. Moreover, implementations for three different systems are shown to demonstrate flexibility of the proposed platform.

Keywords: particle filter, Monte Carlo method, parallel processing, OpenCL, FPGA

1. Introduction

Particle filter, which includes bootstrap filter [1], Monte Carlo filter [2], and CONDENSATION [3], is one promising method to estimate state distribution for the general state space model and has a wide range of applications such as robotics, financial analysis, and environment simulation [4, 5]. Particle filter has a great advantage of being able to handle any functional nonlinearity, and system or measurement noise of any distribution. However, particle filter requires a large computational amount because it uses many candidate vectors called "particles" to approximate the state distribution. This problem prevents particle filter from being used for real-time and low-power applications because the large computational amount causes long computational-time and large power-consumption.

One method for solving this problem is an Field-Programmable Gate Array (FPGA) implementation of particle filter. An FPGA is a reconfigurable LSI consisting of programmable logic blocks, programmable interconnects, embedded memory blocks, Digital Signal Processor (DSP) blocks, etc., and is used to generate custom processors. Performance per power is a great advantage of FPGAs over CPUs. However, traditional FPGA development is inefficient because FPGA designers have to create cycle-by-cycle descriptions of processors using a Hardware Description Language (HDL).

For high-speed and low-power, FPGA implementations of particle-filter-based algorithms have been proposed and are used for various applications such as air traffic management [6], multi-target tracking [7] and tracking current dipole sources of neural activity [8]. In these FPGA implementations, operations for different particles are processed in a pipeline manner. For speeding up further, a parallel implementation with multiple processing elements is proposed, where each processing element processes a fraction of the total number of particles [9]. However, in the simple parallel implementation, exchanging data among processing elements become a bottleneck. Moreover, to enhance the flexibility, some parameterizable FPGA implementations have been proposed and are easy to change their parameters such as the number of particles, image size, and object size [10]. However, changing the FPGA design for different applications, i.e., state space models, requires much design effort in the HDL-based design even though particle filter is flexible.

This paper proposes an FPGA platform for particle filter that is easy to change its design for various applications by utilizing a design tool based on Open Computing Language (OpenCL). OpenCL is a unified parallel programing model on heterogeneous systems consisting of Central Processing Units (CPUs), Graphics Processing Units (GPUs), FPGAs, etc. [11]. As an OpenCL-based design tool for FPGAs, we use the Altera SDK for OpenCL (AOCL) [12] provided from Altera corp. Since the AOCL allows us to use a C-based high-level programming language, we can efficiently design the FPGA. Thanks to the AOCL, users can change state space models, noise distributions and parameters such as the total

number of particles and the number of particles processed in parallel, with a small amount of design effort.

The number of particles processed in parallel determines the performance of the platform. For the scalability of the platform, this paper proposes the OpenCL-oriented parallel implementation of particle filter. There are three major processes in particle filter, which are prediction, weight evaluation and resampling processes. The prediction and weight evaluation processes for multiple particles can be performed in parallel naturally. However, the resampling process becomes a bottleneck because there is no natural parallelism among its operations for different particles [13]. In the AOCL, kernels, which perform the processes, are implemented in a pipeline manner. Hence, this paper proposes the pipeline-oriented parallel implementation of the resampling process.

The implementation results demonstrate the FPGA implementation based on this platform is 106 times faster than the CPU implementation, and the power-delay product of the FPGA implementation is 1.1% of the CPU implementation. The proposed platform makes it possible to use particle filter for real-time and low-power applications.

2. Particle Filter

2.1. State Space Model

Particle filter is the method to estimate the state distribution of a dynamical system from a sequence of observation. The state of the system and observation at the time step k are denoted by vectors $\boldsymbol{x}_k$ and $\boldsymbol{y}_k$ respectively, where k is an integer of the discrete time index. Vectors $\boldsymbol{x}_k$ and $\boldsymbol{y}_k$ are called "state vector" and "observation vector" respectively and are given by

$$\boldsymbol{x}_k = [x_1(k), x_2(k), \ldots, x_n(k)]^T,$$
$$\boldsymbol{y}_k = [y_1(k), y_2(k), \ldots, y_m(k)]^T,$$

where n and m denote the dimensions of the state vector and observation vector respectively. The state vector is not measured directly, while the observation vector is measured. The target system is modeled in a state space representation by using two probability distributions

$$\boldsymbol{x}_k \sim f(\boldsymbol{x}_k|\boldsymbol{x}_{k-1}), \quad \ldots\ldots\ldots\ldots\ldots\ldots \quad (1)$$
$$\boldsymbol{y}_k \sim h(\boldsymbol{y}_k|\boldsymbol{x}_k), \quad \ldots\ldots\ldots\ldots\ldots\ldots \quad (2)$$

where Eqs. (1) and (2) are called "system model" and "observation model" respectively. The system model defines time evolution of the state vector and is written in a conditional probability distribution of a state vector with given the state vector at the previous time step $(k-1)$. Note that, when $k=0$, the initial distribution of the state vector is defined by a probability distribution $\boldsymbol{x}_0 \sim p_0(\boldsymbol{x})$. The observation model defines the relation between the state vector and observation vector, and is written in a conditional probability distribution of the observation vector with given the state vector of the same time step k.

2.2. Estimation Algorithm

Particle filter is one efficient method to estimate the posterior distribution of the state vector at the current time step by approximating the distribution with many particles. Let $x_k^{(i)}$ be a particle at the time step k, where i is an index of the particle. **Fig. 1** illustrates a flowchart of the particle filter algorithm. Note that our platform is based on the most simple particle filter [1, 2]. We explain processes of the flowchart as follows.

(P1) Initialization: Generate initial particles according to the initial distribution such that

$$\boldsymbol{x}_0^{(i)} \sim p_0(\boldsymbol{x}),$$

for $i = 1, 2, \ldots, N$, where N denotes the number of particles.

(P2) Prediction: According to the system model given by Eq. (1), draw temporary particles at the time step k from the particles at the previous time step $(k-1)$ such that

$$\widetilde{\boldsymbol{x}}_k^{(i)} \sim f\left(\boldsymbol{x}_k|\boldsymbol{x}_{k-1}^{(i)}\right),$$

for $i = 1, 2, \ldots, N$.

(P3) Weight evaluation: On receipt of the observation vector, evaluate the weight of each particle according to the observation model given by Eq. (2) such that

$$\widetilde{w}_k^{(i)} \propto h\left(\boldsymbol{y}_k|\widetilde{\boldsymbol{x}}_k^{(i)}\right),$$

for $i = 1, 2, \ldots, N$, where each value of the weight is non-negative.

(P4) Resampling: Draw N particles according to the probability proportional to the weight values such that

$$\boldsymbol{x}_k \sim \begin{cases} \widetilde{\boldsymbol{x}}_k^{(1)} & \text{with probability } \dfrac{\widetilde{w}_k^{(1)}}{\sum_{i=1}^{N} \widetilde{w}_k^{(i)}}, \\ \widetilde{\boldsymbol{x}}_k^{(2)} & \text{with probability } \dfrac{\widetilde{w}_k^{(2)}}{\sum_{i=1}^{N} \widetilde{w}_k^{(i)}}, \\ \vdots & \vdots \\ \widetilde{\boldsymbol{x}}_k^{(N)} & \text{with probability } \dfrac{\widetilde{w}_k^{(N)}}{\sum_{i=1}^{N} \widetilde{w}_k^{(i)}}. \end{cases}$$

This task is executed through three steps as follows. First, from the weight values, compute the cumulative distribution function at the time step k which is given by

$$C_k(i) = \sum_{j=1}^{i} \widetilde{w}_k^{(j)}, \quad \ldots\ldots\ldots\ldots\ldots\ldots \quad (3)$$

for $i = 1, 2, \ldots, N$. Second, select an index $r^{(i)}$ of the drawn particle by using a random number u_i from the uniform distribution over $(0, 1]$. Index $r^{(i)}$ is an integer satis-

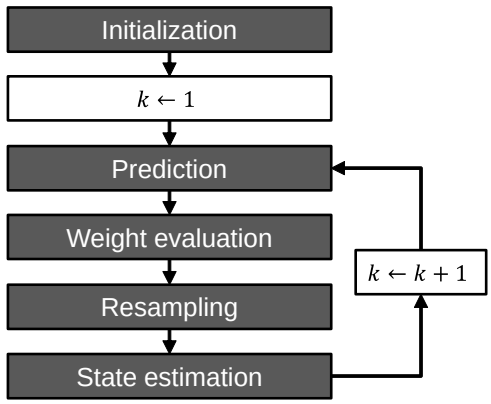


Fig. 1. Flowchart of the particle filter algorithm.

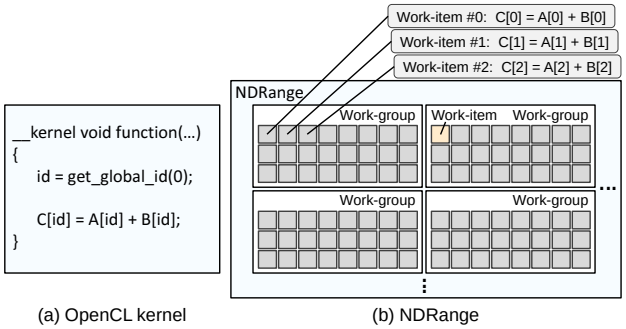


Fig. 2. OpenCL execution model.

fying the equation

$$C_k\left(r^{(i)}-1\right) < u_i C_k(N) \leq C_k\left(r^{(i)}\right), \quad \ldots \quad (4)$$

where $C_k(0)=0$. This step is repeated for $i=1,2,\ldots,N$. Note that, for normalization, the random number is multiplied by the sum $C_k(N)$ of weight values, which is a technique for the FPGA implementation as described in Section 3.2.3. Finally, on the basis of the index $r^{(i)}$, draw posterior particles such that

$$\boldsymbol{x}_k^{(i)} := \widetilde{\boldsymbol{x}}_k^{\left(r^{(i)}\right)},$$

for $i=1,2,\ldots,N$. After the resampling, the weight values of all particles are uniform.

(P5) State estimation: The drawn particles approximate the posterior distribution of the state vector at the time step k. Let $p(\boldsymbol{x}_k|\boldsymbol{y}_{1:k})$ be the posterior distribution, which is given by

$$\left\{\boldsymbol{x}_k^{(i)}\right\}_{i=1}^{N} \approx p(\boldsymbol{x}_k|\boldsymbol{y}_{1:k}),$$

where $\boldsymbol{y}_{1:k}$ denotes the set $\{\boldsymbol{y}_1,\boldsymbol{y}_2,\ldots,\boldsymbol{y}_k\}$. In practical applications such as object tracking and localization, it is necessary to obtain one representative state vector from all particles. One simple method to obtain the representative state vector $\boldsymbol{x}_k$ is given by

$$\boldsymbol{x}_k = \frac{1}{N}\sum_{i=1}^{N}\boldsymbol{x}_k^{(i)}. \quad \ldots \quad (5)$$

3. FPGA Implementation

3.1. Programming Model of Altera SDK for OpenCL

We introduce how an OpenCL program is executed [11, 14]. An OpenCL program is a collection of kernels, each of which is a function in the program code. When a kernel is executed, an index space is defined as shown in **Fig. 2**. This index space is called an N-Dimensional Range (NDRange), and a point of the NDRange is called a work-item. Each work-item executes the same kernel. Each work-item is given a unique ID that is called a global ID. According to the global ID, each set of data is mapped to a work-item. A set of work-item is a work-group and a set of work-group is an NDRange. Work-items of one work-group can share data.

Kernels of the AOCL are classified into two types: "NDRange kernels" and "Single Work-Item kernels." The NDRange kernels utilize work-item-level parallelism. An NDRange kernel has multiple work-items and executes them in parallel like GPU programing models. The NDRange kernels are suitable for the case when work-items do not have memory dependencies. A single work-item kernel is a kernel executed with one work-group containing a single work-item. The single work-item kernels utilize loop-level parallelism and loop-iterations run in parallel. Hence, the single work-item kernel is adapted to the case of data-dependencies arise when loop-iterations run. In the following, we call a loop-iteration a Loop-Iteration-Thread (LI-Thread).

Figure 3 illustrates how the Altera Offline Compiler (AOC) generates custom hardware from OpenCL codes of both an NDRange kernel and a single work-item kernel. Note that, although this figure shows execution of each work-item of the NDRange kernel, each LI-thread of the single work-item kernel is executed in the same way. Operations of the kernel are directly converted to the circuit; the AOC translates an addition to an adder and a subtraction to a subtracter. In addition, the AOC generates the custom pipeline for speeding up computation. At each clock cycle, one work-item is fed into the pipeline and multiple work-items are executed in parallel. Since every work-item is executed in the same pipeline, the pipeline is more efficient in hardware resources than replicating hardware for each work-item. Moreover, the pipeline makes it possible to efficiently use the memory bandwidth because each work-item loads its input data at each clock cycle sequentially. This is essential for FPGAs because the external memory bandwidths of FPGAs are smaller than those of GPUs.

A major difference between the NDRange kernel and the single work-item kernel is a way to share data. In NDRange kernels, sharing data among work-items is done by writing data processed by each work-item to a shared memory as shown in **Fig. 4(a)**. In single work-item kernels, LI-threads can efficiently share data

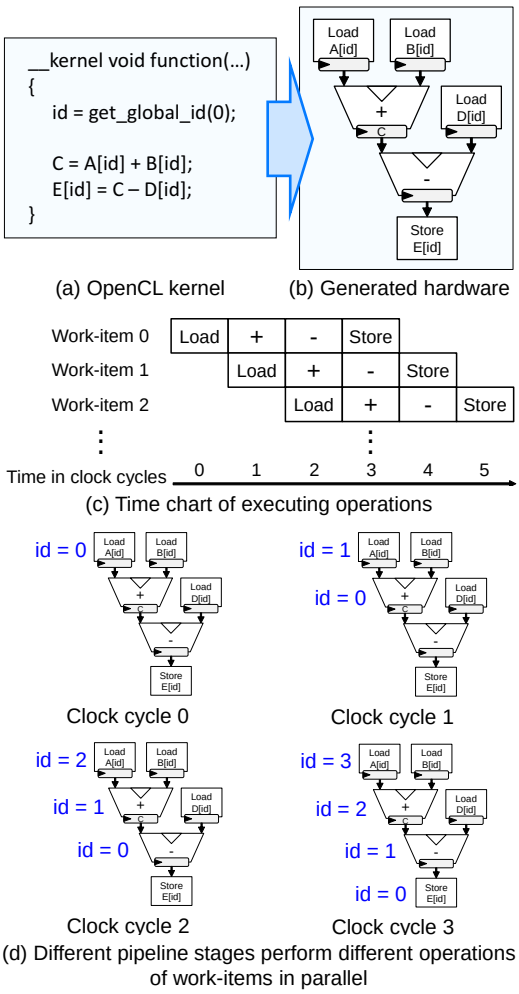


Fig. 3. Generated custom hardware by the AOC [14]. When the AOC finds operations that can be performed in parallel, it generates circuits performing them concurrently.

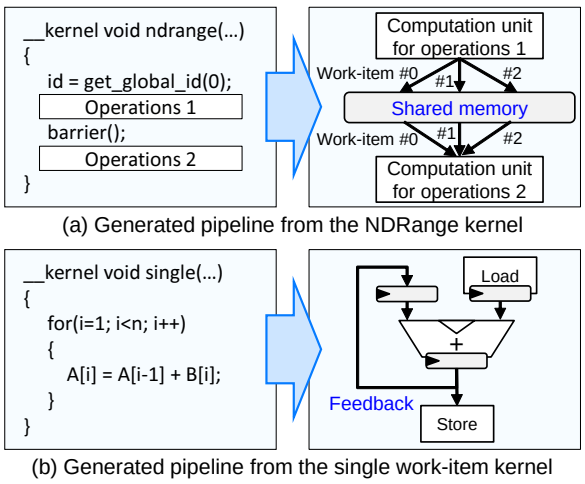


Fig. 4. Difference of the data sharing method between the NDRange kernel and single work-item kernel [15].

through feedbacks without the shared memory as shown in **Fig. 4(b)**. This efficient data sharing of the single work-item kernel is an advantage of the AOCL over typical GPU programing models.

OpenCL handles four memory regions, which are global, constant, local and private memories. A typical FPGA board supporting OpenCL has external SDRAM, on-chip memory blocks and on-chip registers. In the AOCL, the global, constant, local and private memories respectively correspond to SDRAM, SDRAM, memory blocks and registers. A recent high-end FPGA like Stratix V FPGAs has a large memory blocks of 50M bits and about one million of registers. One of the keys to success of the FPGA design is to fully reuse data that are once retrieved from the external memory by using the large on-chip memories. Features of the memory regions are summarized as follows. The global memory has a large capacity, and hence processing data such as images are stored in it. However, the global memory requires high access latency and its bandwidth is small. The constant memory is implemented in SDRAM like the global memory, whereas the constant memory is a read-only memory. Since the constant memory is loaded into an on-chip cache at runtime, it is used for constant data requiring the large bandwidth. The local memory can be randomly accessed without any performance penalty unlike the global memory, and hence a data array requiring complex addressing is stored in the local memory. In addition, the local memory has lower access latency and the larger bandwidth than the global memory. The private memory is mainly implemented in registers. Using the private memory is almost always better than using other memory regions because it provides the larger bandwidth than other memory regions. However, for efficiently using the private memory, an data array in the private memory requires static addressing to determine the access address at compilation time. Otherwise, the private memory may be implemented in the on-chip memory blocks that have the smaller bandwidth than registers. Memory access efficiency determines the maximum performance of a kernel. The bandwidths of the constant, local and private memories are larger than that of the global memory. Hence, minimizing global memory accesses by using other memory regions often leads to improving the kernel performance.

3.2. FPGA Implementation of Particle Filter

3.2.1. Overall Structure

Figure 5 illustrates the kernel structure of the implementation. The implementation of particle filter consists of two parts: (A) Random number generation and (B) Particle filter execution. For the random number generation, we use the Mersenne Twister algorithm [16]. The random number generation is decomposed into two kernels: (A1) Initialization and (A2) Generation. The initialization kernel generates initial states used for generating random numbers and the generation kernel generates random numbers. The particle filter execution is decomposed into four kernels: (B1) Prediction, (B2) Weight evaluation, (B3) Resampling and (B4) State estimation.

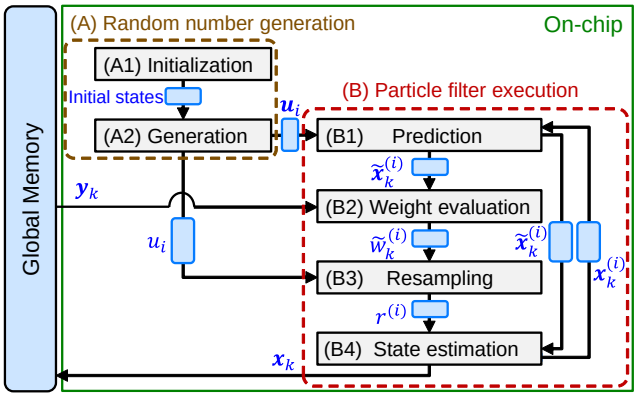


Fig. 5. Overall kernel structure of the implementation.

These kernels perform the processes (P1)–(P5) described in Section 2.2; the prediction, weight evaluation, resampling, and state estimation kernels respectively perform the processes (P1) and (P2), (P3), (P4), and (P5). Note that, since particles are used for both drawing posterior particles and state estimation, drawing posterior particles is performed by the state estimation kernel. Hence, the resampling kernel performs to compute the cumulative distribution function given in Eq.(3) and to select indexes of drawn particles.

This implementation performs the particle filter algorithm at each time step. At each time step, input and output data are transferred between the host PC and FPGA board. The input and output at the time step k are defined as follows.

Input: Observation vector $\boldsymbol{y}_k$
Output: Representative state vector $\boldsymbol{x}_k$

The input data is transferred from the host memory to the global memory before processing and the output data is transferred from the global memory to the host memory after processing.

3.2.2. Basic Ideas for the Kernel Design

The keys to success of the FPGA implementation are as follows.

(i) Minimizing global memory accesses: Since the global memory bandwidth determines the maximum performance of kernels, global memory accesses are minimized whenever possible by utilizing on-chip memories.

(ii) Pipeline-oriented processing: The input data is processed in the incoming order for reducing hardware resources storing the input data.

For efficient parallel processing, each process needs to be broken down to sub-processes that can be performed in parallel. One sub-process is executed by one work-item of an NDRange kernel or one LI-thread of a single work-item kernel. In the following, a work-item or an LI-thread is called a thread. In the algorithm, the operation for each particle can be performed independently except the resampling and state estimation processes. On the basis of this consideration, we assign some particles to one thread, and one thread performs operations for the assigned particles in parallel. In the resampling and state estimation processes, by using single work-item kernels, these processes are executed with efficient data sharing among threads.

3.2.3. Design of Kernels

We describe the fundamental design of kernels. As an example, we use the simple state space model given by

$$x_k = x_{k-1} + v_k, \quad \ldots\ldots\ldots\ldots\ldots\ldots (6)$$
$$y_k = x_k + w_k, \quad \ldots\ldots\ldots\ldots\ldots\ldots (7)$$

where v_k and w_k are zero-means Gaussian white noises with variances 0.01 and 0.09, respectively. The weight value of each particle is given by

$$w_k = g(y_k - x_k), \quad \ldots\ldots\ldots\ldots\ldots\ldots (8)$$

where $g()$ denotes a zero-means Gaussian probability density function with the variance 0.09. The initial distribution $p_0(x)$ is given by a Gaussian distribution $N(0,1)$. The estimated state is the representative state vector given by Eq. (5).

Figure 5 shows the overall kernel structure. In the following, we describe design of kernels.

(A) Random number generation

We describe two kernels performing the random number generation. Note that we implement these kernels based on the OpenCL code of the Mersenne Twister [17] provided from Alter corp.

(A1) Initialization

This kernel generates initial states used for generating random numbers. The initial states are computed by updating a seed value sequentially. Hence, we design this kernel as a single work-item kernel for efficient data sharing among LI-threads. We assign one initial state to one LI-thread. Note that this kernel is executed once when processing of particle filter is started.

(A2) Generation

This kernel generates random numbers from the uniform distribution over $(0,1]$. This operation requires to update states sequentially like the initialization kernel. Hence, we design this kernel as a single work-item kernel. Let R be the number of random numbers used for one particle, and p be the number of particles processed by one thread in the kernels of particle filter execution. In the system model given by Eq. (6), the one random number is enough for one particle, and $R = 1$. One LI-thread generates $(R \times p)$ random numbers and passes them to the prediction kernel. In order to save hardware resources, one of the random numbers is reused for the resampling kernel. Note that, since this kernel does not have to receive data from the host, this kernel runs at all time steps of estimation.

(B) Particle filter execution

We describe four kernels performing particle filter at each time step. In these kernels, one thread performs operations for p particles. In the following, number p is called a parallel number.

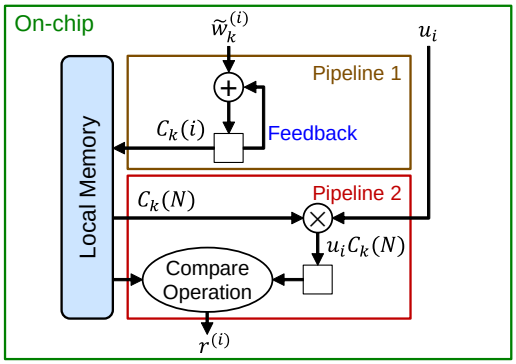


Fig. 6. Operation of the resampling kernel.

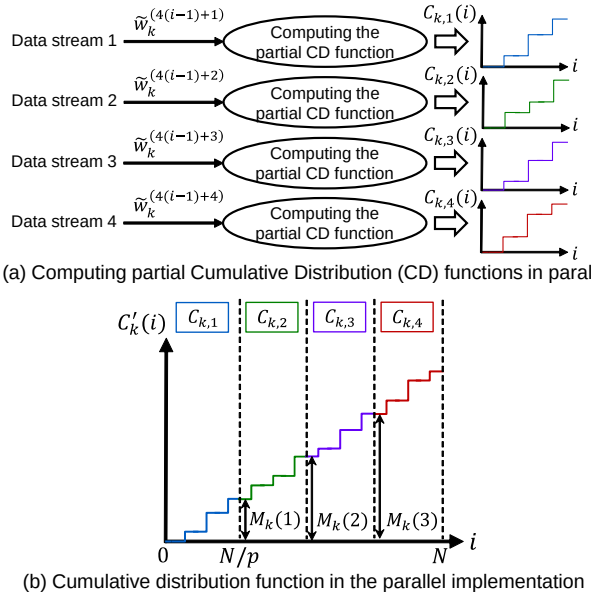


Fig. 7. Computing the cumulative distribution function in the parallel implementation, where the parallel number p is 4.

(B1) Prediction

This kernel generates initial particles or calculates temporary particles at a next time step. Since operations for particles are independent of each other, we design this kernel as an NDRange kernel. For the first time step $k = 1$, each work-item computes initial particles from the initial distribution $N(0,1)$ by using the random numbers. For the other time step $k \neq 1$, each work-item computes temporary particles from Eq. (6). In these operations, we use the Box-Muller transform for generating normally distributed random numbers from uniformly distributed random numbers. The result of each work-item is passed to the weight evaluation and state estimation kernels.

(B2) Weight evaluation

Since operations for particles are independent of each other, we design this kernel as an NDRange kernel. Each work-item loads the observation vector from the global memory and evaluates the weight of particles from Eq. (8). The result of each work-item is passed to the resampling kernel.

For reducing hardware resources, we represent the weight value as a unsigned fixed-point 32-bit integer. For this technique, the weight values given by Eq. (8) are normalized to satisfy the equation $\sup\{\widetilde{w}_k^{(i)}\}_{i=1}^{N} \leq 1$, i.e., the upper bound of the weight values are 1. Note that, to avoid the integer overflow, when the cumulative distribution function given by Eq. (3) is computed, a weight value of one particle is represented as the value that is less than $2^{32-\log_2 N}$.

(B3) Resampling

For efficient data sharing, we design this kernel as a single work-item kernel. Selecting indexes of drawn particles must be started after computing the cumulative distribution function is finished. Hence, two pipelines are generated for computing the cumulative distribution function and selecting indexes respectively, where each pipeline corresponds to one loop statement in the OpenCL code.

Figure 6 illustrates the operation of this kernel. First, each LI-thread computes the cumulative distribution function from Eq. (3) by cumulatively adding the weight value in the incoming order. Although the size of the results is large because the number of particles is many, the results are stored in the local memory for high-speed access. Second, selecting indexes is performed. Note that, before LI-threads compare the random numbers and cumulative distribution function, the value of each random number is multiplied by $C(N)$ for normalization. Since the sum $C(N)$ of all weight values is obtained after computing the cumulative distribution function is finished, to normalize the random number is more efficient in hardware resources than to normalize the cumulative distribution function. Each LI-thread selects the index satisfying Eq. (4) using the binary search technique for efficient search. Finally, the selected indexes are passed to the state estimation kernel.

The resampling process is performed in parallel for multiple particles. Selecting indexes can be performed in parallel naturally by using random numbers. However, computing the cumulative distribution function become a problem because there is no natural parallelism among its operations for different particles. The cumulative distribution function is computed as follows. First, partial cumulative distribution functions are computed in parallel for p data streams from the weight evaluation kernel as shown in **Fig. 7(a)**. Let $C_{k,l}(i)$ be a value of the partial cumulative distribution function in the data stream l and it is given by

$$C_{k,l}(i) = \sum_{j=1}^{i} \widetilde{w}_k^{(p(j-1)+l)},$$

where $i = 1,2,\ldots,(N/p)$ and $l = 1,2,\ldots,p$. Second, last values of the partial cumulative distribution functions (i.e., $C_{k,l}(N/p)$) are added cumulatively as follows:

$$M_k(i) = \sum_{j=1}^{i} C_{k,j}\left(\frac{N}{p}\right),$$

where $i = 1, 2, \ldots, (p-1)$. Although this operation causes the overhead for the parallel implementation and requires $(p-1)$ sequential additions, this overhead is relatively small because $(p-1)$ is typically much smaller than (N/p). Finally, the cumulative distribution function is given by

$$C'_k(i) = C_{k,\left(\left\lfloor \frac{i-1}{\left(\frac{N}{p}\right)} \right\rfloor + 1\right)} \left(\left((i-1) \bmod \frac{N}{p} \right) + 1 \right) + M_k \left(\left\lfloor \frac{i-1}{\left(\frac{N}{p}\right)} \right\rfloor \right), \quad \ldots\ldots\ldots (9)$$

where $i = 1, 2, \ldots, N$, $M_k(0) = 0$ and $\lfloor x \rfloor$ denotes the maximum integer that does not exceed the number x. **Fig. 7 (b)** illustrates the cumulative distribution function computed from Eq. (9). Each section of this function whose length is (N/p) corresponds to the partial cumulative distribution function and $M_k(l)$ is the offset value for $C_{k,l+1}(i)$. One element of the cumulative distribution function can be computed from Eq. (9) without computing all elements of that.

In the parallel implementation, selecting indexes is performed by using the cumulative distribution function computed from Eq. (9). Let i_{para} be a selected index by using the function computed from Eq. (9). By using i_{para}, the index $r^{(i)}$ of the drawn particle described in Section 2.2 is given by

$$r^{(i)} = p \left((i_{para} - 1) \bmod \frac{N}{p} \right) + \left\lfloor \frac{i_{para} - 1}{\left(\frac{N}{p}\right)} \right\rfloor + 1. \quad \ldots\ldots\ldots\ldots\ldots\ldots (10)$$

When the selected index i_{para} is given, we compute the index $r^{(i)}$ of the drawn particle from Eq. (10).

(B4) State estimation

This kernel executes drawing posterior particles and computing the representative state vector. For efficient data sharing, we design this kernel as single work-item kernel. Drawing particles must be started after all temporary particles are received because temporary particles used are determined at runtime. Hence, two pipelines are generated for storing temporary particles and drawing posterior particles, respectively.

Figure 8 illustrates the operation of this kernel. First, each LI-thread stores all temporary particles in the local memory for high-speed access. Second, drawing particles and computing the representative state vector are executed. Each LI-thread receives the selected indexes from the resampling kernel and draws posterior particles corresponding to the indexes from the local memory. The drawn particles are used for computing the sum of all posterior particles and are passed to the prediction kernel. After computing the sum, the representative state vector is computed by multiplying the sums by $1/N$. The represented vector is stored in the global memory.

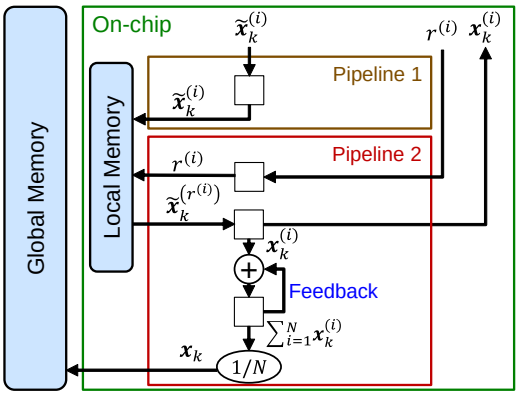


Fig. 8. Operation of the state estimation kernel.

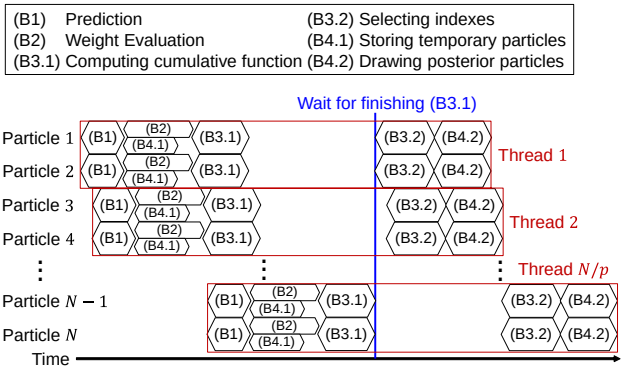


Fig. 9. Timechart of the particle filter execution kernels, where the parallel number p is 2.

3.2.4. Connecting Between Kernels

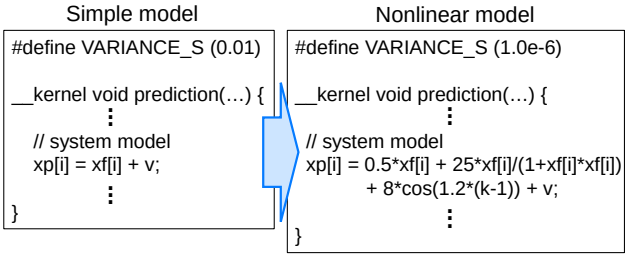
The AOCL provides a mechanism enabling to connect different kernels directly through an on-chip FIFO buffer, called a "channel," with high-efficiency and low-latency [18]. The kernels are connected by channels as shown in **Fig. 5**, where a channel is represented by a rectangle on the line connecting kernels. The use of channels has following advantages.

(i) Saving the global memory bandwidth:

In the typical OpenCL, which includes its version 1.0, 1.1 and 1.2, the global memory must be used for data transfer between kernels. By using channels, results of each kernel can be passed to the next kernel without using the global memory.

(ii) Multiple kernels executed in parallel:

Since kernels pass their results synchronously by channels, multiple kernels with mutual data-dependencies can be executed in parallel. **Fig. 9** illustrates the timechart of kernels of particle filter execution. A hexagon denotes the processing time of each operation for one particle and an internal symbol of the hexagon denotes the operation. Although the generation kernel is omitted, the generation kernel is executed in parallel with these kernels. Note that, in the resampling kernel, selecting indexes is started after computing the cumulative distribution function. A rectangle denotes the processing time of operations performed



(a) Changing the system model and variance of the system noise

```
#define N_STS (6)              // dimension of state vector
#define N_OBS (3)              // dimension of observation vector
#define N_PAR (2)              // parallel number
#define N_RND (4*N_PAR)        // the number of random numbers
#define N_PART (8192)          // the number of particles
```

(b) Definition of implementation parameters

Fig. 10. Flexibility of the implementation.

by one thread. The computation is speeded up by utilizing both pipeline processing and spatial parallel processing.
(iii) Improving development efficiency:

Processes with mutual data-dependencies can be implemented as multiple kernels while keeping their performance. Hence, the use of channels improves development efficiency and maintainability.

3.2.5. Flexibility of Designs

The AOCL allows us to use the high-level programming language for the FPGA design. Hence, the OpenCL-based design is much easier to change than the HDL-based design. For example, let us change the simple model given by Eqs. (6) and (7) to the nonlinear model given by

$$x_k = 0.5x_{k-1} + \frac{25x_{k-1}}{1+x_{k-1}^2} + 8\cos(1.2(k-1)) + v_k, \quad \ldots\ldots\ldots \quad (11)$$

$$y_k = \frac{x_k^2}{20} + w_k, \quad \ldots\ldots\ldots\ldots\ldots \quad (12)$$

where v_k and w_k are zero-means Gaussian white noises with variances $1e^{-6}$ and 0.01, respectively. The initial distribution and way to compute the representative state vector are not change. In this case, the prediction kernel is changed as shown in **Fig. 10(a)** and the weight evaluation kernel is changed in the same way. The variances of the noises v_k and w_k are also changed. By only these changes, we can change the state space model easily. Parameters of the implementation can be also easily changed by changing "#define" statements as shown in **Fig. 10(b)**.

4. Evaluation

4.1. Performance Comparison Between CPU and FPGA Implementations

We compare performance between CPU and FPGA implementations. As the state space model, the simple model given by Eqs. (6) and (7) is used. The number N of particles is determined as 8192, and the parallel number p in the FPGA implementation is determined as 1. As a CPU, we use Core i7-3960X from Intel corp. running at a clock frequency of 3.3 GHz–3.9 GHz. The CPU implementation is the simple implementation with a single thread. For the CPU implementation, we use the C/C++ Optimizing Compiler Version 18 for x64 from Microsoft corp. The CPU is also used for the host in the FPGA implementation. As a FPGA board, we use the DE5-Net board from Terasic corp. that has the Stratix V A7 FPGA from Altera corp. For the FPGA implementation, we use the Altera SDK for OpenCL, Version 15.0.2.

Table 1. Processing time and power-delay (PD) product of CPU and FPGA implementations.

	Corei7-3960X (1-thread)	DE5-Net FPGA board
Processing time (ms)	15.13	0.14
PD product (W×s)	0.86	0.0095

For evaluation metrics, we use the processing time and power-delay product. The processing time is the time for executing particle filter at one time step, and is measured as the average time of 1000 executions. In the FPGA implementation, the processing time includes the time of data transfer between the host PC and FPGA board. The power-delay product, which represents energy-efficiency, is defined as the product of the processing time and power consumption. The power consumption is measured with a power meter (HIOKI AC/DC POWER HiTESTER 3334). Note that the power consumption is the difference between the powers of a whole computer in the idle state and operation state. **Table 1** summarizes the processing time and power-delay product of each implementation. The FPGA implementation is 106 times faster than the CPU implementation. Moreover, the power-delay product of the FPGA implementation is 1.1% of the CPU implementation. The results demonstrate that the FPGA implementation is much faster and more energy-efficient than the CPU implementation.

4.2. Implementations for Different State Space Models

In order to demonstrate flexibility of the proposed platform, we evaluate implementations for three state space models as follows. The first one is the simple model given by Eqs. (6) and (7). The implementation using the simple model is called $\mathbf{I}_{simple}$ in the following, which is the same implementation in Section 4.1. The second one is the nonlinear model given by Eqs. (11) and (12), where the initial distribution is given by a Gaussian distribution $N(0,25)$. The implementation using the nonlinear model is called $\mathbf{I}_{nonlinear}$ in the following. The third one is the model used for color tracking. The system model is given by

$$\boldsymbol{x}_k = \boldsymbol{F}\boldsymbol{x}_{k-1} + \boldsymbol{G}\boldsymbol{v}_k,$$

Table 2. Implementation results of FPGA implementations for three models.

	Stratix V A7	$\mathbf{I}_{simple}$	$\mathbf{I}_{nonlinear}$	$\mathbf{I}_{colortrack}$	$\mathbf{I}_{simple}$
Logic[a]	234,720	54,748 (23%)	58,552 (25%)	162,580 (69%)	55,474 (24%)
Registers	938,880	89,007 (9%)	97,752 (10%)	209,759 (22%)	90,060 (10%)
RAM blocks	2,560	444 (17%)	449 (18%)	1,775 (69%)	1,582 (62%)
DSP blocks	256	26 (10%)	36 (14%)	38 (15%)	26 (10%)
Frequency (MHz)		298	288	166	188
The number of particles N		8192	8192	8192	65536
Parallel number p		1	1	1	1
Processing time (ms)		0.14	0.15	4.05	0.86
Standard deviation (ms)		0.0082	0.0086	0.16	0.039

a. Logic denotes Adaptive Logic Modules (ALMs).

where $\boldsymbol{x}_k = [x_k, y_k, x_k^p, y_k^p, b_k, h_k]^T$, $\boldsymbol{v}_k = [v_k^x, v_k^y, v_k^b, v_k^h]^T$,

$$\boldsymbol{F} = \begin{bmatrix} 2 & 0 & -1 & 0 & 0 & 0 \\ 0 & 2 & 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

and

$$\boldsymbol{G} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Here x_k and y_k denote coordinates of the current target position; numbers x_k^p and y_k^p denote coordinates of the previous target position; numbers b_k and h_k denote the size of the image window centered on the coordinate (x_k, y_k) and are used for weight evaluation. Number $v_k^x \sim N(0,25)$, $v_k^y \sim N(0,25)$, $v_k^b \sim N(0,4)$ and $v_k^h \sim N(0,4)$. The weight value of each particle is given by

$$w_k = \frac{T}{W},$$

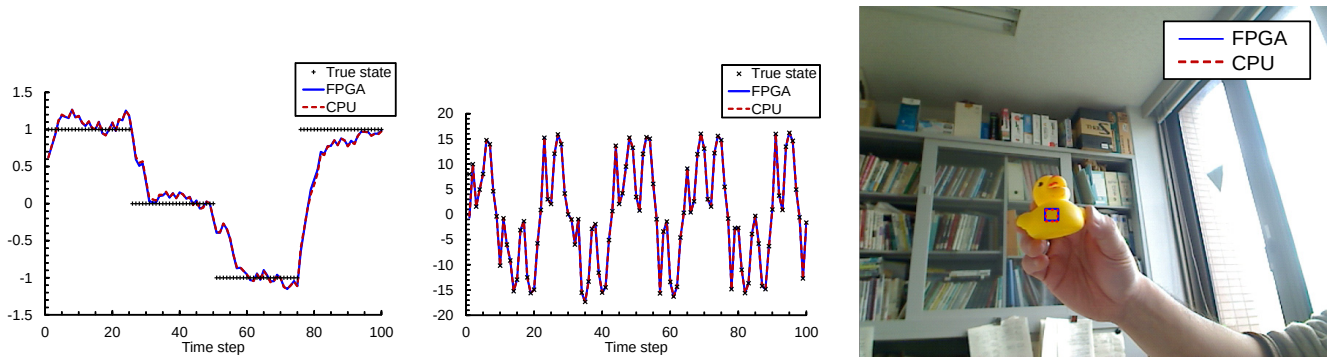
where T denotes the number of pixels of the target color in the image window and W denotes the number of all pixels in the image window. The maximum size of the image window is determined as 50×50 pixels. Note that color tracking is executed both the host and FPGA serially; the host performs the operation of creating the mask image whose pixel values of the target color pixels are 1 and other pixel values are 0, and after that, the FPGA performs all the other operations. The implementation using the this model is called $\mathbf{I}_{colortrack}$ in the following.

The implementation results are summarized in **Table 2**, which contains the resource utilization, the maximum frequency, the number of particles, the parallel number, the processing time and the standard deviation of the processing time. In these implementations, the parallel number is determined as 1. The "Stratix V A7" column shows the specification of the FPGA (Stratix V A7) [19]. The processing time is measured in the same way as described in Section 4.1. Note that the processing time of $\mathbf{I}_{colortrack}$ includes the time required for the host to create the mask image; the time for creating the mask image is 3.63 (ms) and the time of processing all other operations is 0.42 (ms). The standard deviation is calculated from the processing time of 1000 executions. The standard deviation of each implementation is much smaller than its processing time. The right-most column shows the implementation results of $\mathbf{I}_{simple}$ for the case when the number of particles are maximized; the maximum number of particles is 65536 under the resource constraint of the used FPGA. The performances of these implementations are sufficient for typical real-time applications. **Fig. 11** illustrates estimation results for the three models by CPU and FPGA implementations with 8192 particles. The cross marks, dashed lines and solid lines represent true states, results of the CPU implementation and results of the FPGA implementation, respectively. The size of the image used for $\mathbf{I}_{colortrack}$ is 640×480 pixels. These results demonstrate that the results of CPU and FPGA implementations are almost the same.

5. Discussion for High-Speed Implementations

The proposed implementation can be more speeded up. The implementation performs the particle filter algorithm at each time step since the output and input data are transferred between the host PC and FPGA board at each time step. However, if the data transfer is not required, operations at different time steps can be executed in parallel. We present two types of high-speed implementations based on this idea as follows.

The first one is based on overlapping operations without replicating the local memory. We call this implementation $\mathbf{I}_{HS1}$. **Fig. 12(a)** illustrates the timechart of the this implementation. In the resampling kernel, the operations of both computing the cumulative distribution function and selecting indexes use the same local memory storing the function. For this reason, the operation of computing the cumulative distribution function at the time step k must be started after the operation of selecting indexes for all particles at the previous time step $(k-1)$ is finished; in **Fig. 12(a)**, the operation of (B3.1) at the time step k must be started after the operation of (B3.2) for all particles at



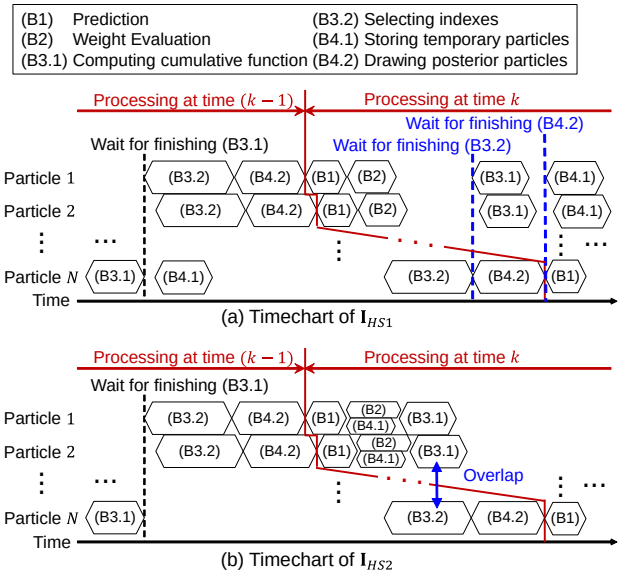
(a) Estimation result of $\mathbf{I}_{simple}$ (b) Estimation result of $\mathbf{I}_{nonlinear}$ (c) Estimation result of $\mathbf{I}_{colortrack}$

Fig. 11. Estimation results of CPU and FPGA implementations.

the previous time step $(k-1)$ is finished. Besides, in the state estimation kernel, the operations of both storing temporary particles and drawing posterior particles use the same local memory storing the temporary particles. For this reason, the operation of storing temporary particles at the time step k must be started after the operation of drawing posterior particles at the previous time step $(k-1)$ is finished; in **Fig. 12(a)**, the operation of (B4.1) at the time step k must be started after the operation of (B4.2) for all particles at the previous time step $(k-1)$ is finished. Hence, the operations at the time steps $(k-1)$ and k are overlapped based on the above restrictions.

The second one is based on overlapping operations with replicating the local memory. We call this implementation $\mathbf{I}_{HS2}$. **Fig. 12(b)** illustrates the timechart of the this implementation. In the resampling kernel, the memory storing the cumulative distribution function is replicated for starting the operation at the time step k while the operation at the previous time step $(k-1)$ is executed; in **Fig. 12(b)**, the operations of (B3.1) at the time step k is started while the operations (B3.2) at the previous time step $(k-1)$ is executed. Similarly, in the state estimation kernel, the memory storing the temporary particles is replicated for starting the operation at the time step k while the operation at the previous time step $(k-1)$ is executed; in **Fig. 12(b)**, the operations of (B4.1) at the time step k is started while the operations (B4.2) at the previous time step $(k-1)$ is executed. This technique leads to high performance, though utilization of the memory blocks is increased.

We implement $\mathbf{I}_{HS1}$ and $\mathbf{I}_{HS2}$ by using the simple state space model given by Eqs. (6) and (7). The number of particles is 8192. The results of these implementations are summarized in **Table 3**, which includes $\mathbf{I}_{simple}$ for comparison. The total processing time is the time of executing particle filter during time steps 1–1000. For the parallel number of 1, $\mathbf{I}_{HS1}$ is 1.9 times faster than $\mathbf{I}_{simple}$, and $\mathbf{I}_{HS2}$ is 3.6 times faster than $\mathbf{I}_{simple}$. In this case, the memory blocks of $\mathbf{I}_{HS2}$ are increased compared to those of $\mathbf{I}_{HS1}$. Performances of the implementations can be improved by increasing the parallel number. As for $\mathbf{I}_{HS1}$, 3.5-time speedup is achieved by increasing the parallel number from 1 to 4. As for $\mathbf{I}_{HS2}$, 2.6-time speedup is achieved by increasing the parallel number from 1 to 4.



(a) Timechart of $\mathbf{I}_{HS1}$

(b) Timechart of $\mathbf{I}_{HS2}$

Fig. 12. Timechart of high-speed implementations.

6. Conclusion

This paper proposes the FPGA platform for particle filter for the purpose of high-speed and low-power, which can be easily used for various state space models by the OpenCL-based design. The key to speeding up computation is to utilize pipeline processing and spacial parallel processing. The proposed implementation makes it possible to use particle filter for real-time and low-power applications. As future work, we are building a large-scale particle filter accelerator with more than several million particles by connecting multiple FPGAs. We are also planning to develop the FPGA-oriented OpenCL-code generator for the user-specified state space model, noise distribution and parameters such as the number of particles and parallel number.

Table 3. Implementation results of high-speed implementations.

	$\mathbf{I}_{simple}$	$\mathbf{I}_{HS1}$	$\mathbf{I}_{HS2}$	$\mathbf{I}_{HS1}$	$\mathbf{I}_{HS2}$
Logic[a]	54,748 (23%)	58,557 (25%)	58,791 (25%)	83,397 (36%)	83,957 (36%)
Registers	89,007 (9%)	97,718 (10%)	97,923 (10%)	158,362 (17%)	160,465 (17%)
RAM blocks	444 (17%)	486 (19%)	678 (26%)	548 (21%)	1,167 (46%)
DSP blocks	26 (10%)	26 (10%)	26 (10%)	88 (34%)	88 (34%)
Frequency (MHz)	298	220	221	208	165
The number of particles N	8192	8192	8192	8192	8192
Parallel number p	1	1	1	4	4
Total processing time (ms)	142.96	76.74	39.56	21.98	15.09

a. Logic denotes Adaptive Logic Modules (ALMs).

Acknowledgements

This work is supported by MEXT KAKENHI Grant Number 16K12404.

References:

[1] N. J. Gordon, D. J. Salmond, and A. F. M. Smith, "Novel approach to nonlinear/non-Gaussian Bayesian state estimation," IEE Proc. F, Vol.140, No.2, pp. 107-113, 1993.

[2] G. Kitagawa, "Monte Carlo Filter and Smoother for Non-Gaussian Nonlinear State Space Models," J. of Computational and Graphical Statistics, Vol.5, No.1, pp. 1-25, 1996.

[3] M. Isard and A. Blake, "CONDENSATION – Conditional Density Propagation for Visual Tracking," Int. J. of Computer Vision, Vol.29, Issue 1, pp. 5-28, 1998.

[4] N. Ikoma, "Hands and Arms Motion Estimation of a Car Driver with Depth Image Sensor by Using Particle Filter," Proc. of the 14th Int. Symp. on Advanced Intelligent Systems (ISIS), pp. 75-84, 2013.

[5] N. Ikoma and A. Asahara, "Real Time Color Object Tracking on Cell Broadband Engine Using Particle Filters," J. of Advanced Computational Intelligence and Intelligent Informatics (JACIII), Vol.14, No.3, pp. 272-280, 2010.

[6] T. C. P. Chau, J. S. Targett, M. Wijeyasinghe, W. Luk, P. Y. K. Cheung, B. Cope, A. Eele, and J. Maciejowski, "Accelerating Sequential Monte Carlo Method for Real-time Air Traffic Management," ACM SIGARCH Computer Architecture News, Vol.41, Issue 5, pp. 35-40, 2013.

[7] Z. Shi, Y. Liu, S. Hong, J. Chen, and X. S. Shen, "POSE: Design of Hardware-Friendly Particle-Based Observation Selection PHD Filter," IEEE Trans. on Industrial Electronics, Vol.61, No.4, pp. 1944-1956, 2014.

[8] L. Miao, J. J. Zhang, C. Chakrabarti, and A. Papandreou-Suppappola, "Efficient Bayesian Tracking of Multiple Sources of Neural Activity: Algorithms and Real-Time FPGA Implementation," IEEE Trans. on Signal Processing, Vol.61, No.3, pp. 633-647, 2013.

[9] M. Bolić, "Architectures for Efficient Implementation of Particle Filters," Ph.D. thesis, State University of New York at Stony Brook, 2004.

[10] P. Engineer, R. Velmurugan, and S. B. Patkar, "Parameterizable FPGA framework for particle filter based object tracking in video," 28th Int. Conf. on VLSI Design (VLSID), pp. 35-40, 2015.

[11] Khronos Group, 2010, The OpenCL Specification, available: https://www.khronos.org/registry/cl/specs/opencl-1.0.48.pdf.

[12] "Altera SDK for OpenCL," https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html.

[13] A. Athalye, M. Bolić, S. Hong, and P. M. Djurić, "Generic Hardware Architectures for Sampling and Resampling in Particle Filters," EURASIP J. on Applied Signal Processing, Vol.2005, No.17, pp. 2888-2902, 2005.

[14] Altera SDK for OpenCL Best Practices Guide, available: https://www.altera.com/en_US/pdfs/literature/hb/opencl-sdk/aocl-best-practices-guide.pdf.

[15] "OpenCL: Single-threaded (Task) vs Multi-threaded (NDRange)," http://www.altera.com/customertraining/webex/OpenCLKern/player.html.

[16] M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator," ACM Trans. on Modeling and Computer Simulation (TOMACS), Vol.8, Issue 1, pp. 3-30, 1998.

[17] "Monte Carlo Black-Scholes Asian Options Pricing Design Example," https://www.altera.com/support/support-resources/design-examples/design-software/opencl/black-scholes.html.

[18] Altera SDK for OpenCL Programming Guide, available: https://www.altera.com/en_US/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf.

[19] Stratix V Device Overview, available: https://www.altera.com/en_US/pdfs/literature/hb/stratix-v/stx5_51001.pdf.

Name:
Shunsuke Tatsumi

Affiliation:
Graduate School of Information Sciences, Tohoku University

Address:
6-6-05 Aramaki Aza Aoba, Aoba, Sendai 980-8579, Japan

Brief Biographical History:

2012 Received the B.E. degree in information engineering from Tohoku University

2014 Received the M.S. degree in information sciences from Tohoku University

2014- Ph.D. Student in the Graduate School of Information Sciences, Tohoku University

Main Works:

• Reconfigurable computing and image processing

Name:
Masanori Hariyama

Affiliation:
Graduate School of Information Sciences, Tohoku University

Address:
6-6-05 Aramaki Aza Aoba, Aoba, Sendai 980-8579, Japan

Brief Biographical History:
1992 Received the B.E. degree in electronic engineering from Tohoku University
1994 Received the M.S. degree in information sciences from Tohoku University
1997 Received the Ph.D. degree in information sciences from Tohoku University
2003- Associate Professor with the Graduate School of Information Sciences, Tohoku University

Main Works:
- Real-world applications such as robotics
- Medical applications
- Big-data and high-performance computing
- VLSI computing for real-world applications
- Reconfigurable computing

Membership in Academic Societies:
- The Institute of Electronics Information and Communication Engineers (IEICE)
- The Institute of Electrical and Electronics Engineers (IEEE)
- The Robotics Society of Japan (RSJ)
- Information Processing Society of Japan (IPSJ)
- The Society of Instrument and Control Engineers (SICE)

Name:
Norikazu Ikoma

Affiliation:
Faculty of Engineering, Nippon Institute of Technology

Address:
4-1 Gakuendai, Miyashiro-machi, Minamisaitama-gun, Saitama 345-8501, Japan

Brief Biographical History:
1995-1997 Research Associate, Faculty of Information Science, Hiroshima City University
1998-2003 Assistant Professor, Faculty of Engineering, Kyushu Institute of Technology
2003-2015 Associate Professor, Faculty of Engineering, Kyushu Institute of Technology
2016- Professor, Faculty of Engineering, Nippon Institute of Technology

Main Works:
- "Real-Time Face Decorations of Enlarging Eyes and Whitening Skin in Video Based on Face Posture Estimation by Particle Filter," J. of Advanced Computational Intelligence and Intelligent Informatics (JACIII), Vol.17, No.3, pp. 392-403, 2013.
- "Real Time Color Object Tracking on Cell Broadband Engine Using Particle Filters," J. of Advanced Computational Intelligence and Intelligent Informatics (JACIII), Vol.14, No.3, pp. 272-280, 2010.
- "Tracking of Multiple Moving Objects in Dynamic Image of Omni-directional Camera Using PHD Filter," J. of Advanced Computational Intelligence and Intelligent Informatics (JACIII), Vol.12, No.1, pp. 16-25, 2008.

Membership in Academic Societies:
- The Institute of Electrical and Electronics Engineers (IEEE)
- The Japan Society for Fuzzy Theory and Intelligent Informatics (SOFT)
- The Institute of Electronics, Information and Communication Engineers (IEICE)