18th International Conference on Information Fusion
Washington, DC - July 6-9, 2015

GPU-Accelerated Progressive Gaussian Filtering with Applications to Extended Object Tracking

Jannik Steinbring and **Uwe D. Hanebeck**
Intelligent Sensor-Actuator-Systems Laboratory (ISAS)
Institute for Anthropomatics and Robotics
Karlsruhe Institute of Technology (KIT), Germany
jannik.steinbring@kit.edu, uwe.hanebeck@ieee.org

***Abstract*—Since the last years, Graphics Processing Units (GPUs) have massive parallel execution capabilities even for non-graphic related applications. The field of nonlinear state estimation is no exception here. Particle Filters have already been successfully ported to GPUs. In this paper, we propose a GPU-accelerated variant of the Progressive Gaussian Filter (PGF). This allows us to combine the advantages of the particle flow with the ability to process thousands of measurements at once in order to improve state estimation quality. To get a meaningful comparison between its CPU and GPU variants, we additionally propose a likelihood for tracking a sphere and its extent in 3D based on noisy point measurements. The likelihood considers the physical relationship between sensor, measurement, and sphere to best exploit the information of the received measurements. We evaluate the GPU implementation of the PGF using the proposed likelihood in combination with tens of thousands of measurements. Although the CPU implementation fully exploits parallelization techniques such as SSE and OpenMP, the GPU-accelerated PGF reaches speedups over 20 and real-time tracking can nearly be achieved.**

***Keywords*—PGF, GPU, OpenCL, Extended Object Tracking.**

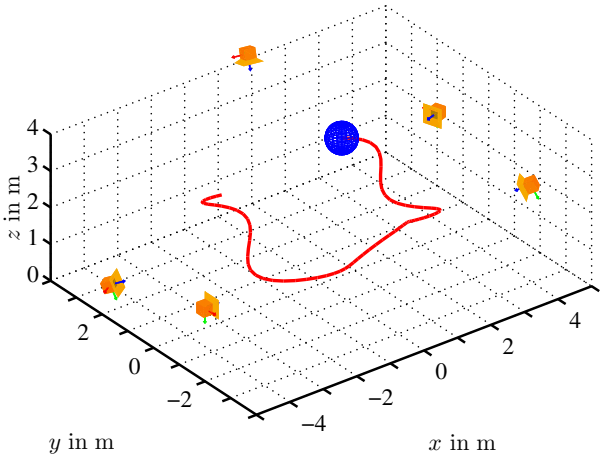

Figure 1: Tracking of a sphere in 3D (blue) using five depth cameras (orange), and part of the sphere's trace (red).

I. Introduction

In recent years increased sensor resolutions have made it possible to observe more and more measurements from a scenario *at a given time*, such as in [1]. Also several (maybe complementary) sensors can observe the same scene, leading to an increased number of measurements that are available for processing (see Fig. 1). Especially the field of extended object tracking benefits from this fact. Objects are now modeled as a (simplified) shape rather than a single point [2]–[4]. Hence, the more measurements are available from the object the better can its pose (and also its shape) be estimated. Usually, this is done in a probabilistic fashion using (nonlinear) state estimation techniques. Here, the object pose, its shape parameters, and maybe other information such as velocities are collected in the system state, whereas the system model is based on the object's kinematic behavior and the measurement model on how measurements are related to the object [5].

In case of linear models corrupted by additive Gaussian noise, the Kalman Filter (KF) is the optimal state estimator in the sense of a minimum mean squared error [6]. Moreover, under the common assumption of mutually independent measurement noise it is valid to process measurements from one time step sequentially by executing several measurement updates in a row. This, of course, without any prediction in between, as the measurements stem from the same time step. Doing so is highly recommended as the computational effort increases cubically for the KF when processing a rising number of measurements at once. Due to the optimality of the KF, the final state estimate is equivalent to the one that would have been obtained by processing all measurements in a single update. However, in most practical scenarios measurement models are nonlinear.

Unfortunately, when applying linear estimators such as the Extended Kalman Filter (EKF) [6], the Unscented Kalman Filter (UKF) [7], or the Smart Sampling Kalman Filter (S^2KF) [8] to such a nonlinear estimation problem, processing measurements sequentially is inadvisable. More precisely, each single measurement update results only in an approximation of the posterior state estimate. Then, when performing many measurement updates sequentially, these errors accumulate and can finally cause a state estimate that is (much) different from the one obtained by processing all measurements in a single update. An additional problem is that, after processing several hundreds of measurements sequentially, the entries of the state covariance will become very small. Consequently, this effect will practically ignore further measurements as the estimator is already very confident about its current state estimate. This fact is especially important when the processed measurements do not represent the current system state correctly, whereas the entire set of available measurements does. As a thought

experiment, consider a set of measurements from a target's surface. When processing these measurements sequentially and the first measurements stem only from one half of the target, the estimator would ignore the measurements from the other half, and hence, interprets this as the target being only half the size of the true one. That is, the order of processing measurements matters[1]. A random rearrangement of the measurements could mitigate this but would still suffer from the problem of ignoring measurements.

A solution to this might be to increase the state covariance after processing a batch of measurements to allow changes when processing further measurements. However, this requires unintuitive and error-prone parameter tuning and does not solve the problem of error accumulation. Another solution would be to reduce the number of measurements by rejecting measurements randomly, but this ignores information about the system state and will not scale to the future when assuming a rise in available measurements.

Hence, in the nonlinear case, it is to be preferred to process all measurements from one time step *at once*. However, as mentioned above, linear filters do not scale well with an increasing number of measurements. In order to overcome this issue, we propose the use of state estimators that directly work with a likelihood function instead. The advantage of using a likelihood is that the complexity of its evaluation increases only linearly with a rising number of measurements[2]. Moreover, estimators relying on likelihoods have the additional benefit of an, in general, superior state estimate compared to linear filters due to the avoidance of linearizing the measurement model in some way.

Popular nonlinear estimators are, for example, Particle Filters (PFs) and their various derivatives [9]–[11]. The downside of Particle Filters is the problem of sample degeneration: too many particles can get lost in case of an unfavorable situation in which the support of the likelihood and the prior state density do not overlap sufficiently. The result is a diminished estimation performance or even filter divergence. As a consequence, a large number of particles may be required to get satisfactory results, but this in turn leads to a higher computational effort. Additionally, the creation of random numbers for prediction and resampling as well as the resampling itself impose further overhead.

A solution to these problems was proposed with the Progressive Gaussian Filter (PGF) [12], [13] which makes use of the so-called particle flow approach. The idea is here to move the particles during a measurement update to the important regions of the state space to circumvent the problem of sample degeneration and simultaneously reduce the required number of samples drastically. Moreover, the PGF varies the number of likelihood evaluations from update to update depending on how much information the new measurements possess. This is another improvement compared to Particle Filters, where the number of likelihood evaluations is constant over time.

Although the PGF improves the measurement update runtime with its particle flow, currently available CPUs are still a limiting factor when it comes to time-critical applications such as real-time object tracking. To accelerate the PGF when processing many measurements, it stands to reason to exploit the computational power of a Graphics Processing Unit (GPU). The large number of measurements makes it possible to fully utilize the GPU, and hence, to achieve good performance improvements. As the evaluation will show, this allows real-time extended object tracking with thousands of measurements. Moreover, by transferring the state estimation to the GPU, the CPU load will be reduced and is available for other tasks, e.g., measurement pre-processing. The PGF runtime improvements are also not restricted to tracking applications. Every task that involves nonlinear state estimation will benefit from the speedup.

Nonetheless, also classical Particle Filters can profit from the capabilities of GPUs. Here, resampling and random number generation are the most challenging parts of porting a PF to the GPU and are also the runtime bottleneck [14]. Additionally, due to the lack of a proper random number generator, in [14] the required random numbers are computed on the CPU. Unfortunately, this caused a large amount of data that had to be transferred to the GPU on every measurement update.

Due to this fact, the authors of [15] implemented a random number generator for the GPU on their own to be able to execute the PF completely on a GPU, and used it for a single target video tracking application. Their GPU version is about ten times faster than their OpenMP version on a multi-core CPU. Moreover, today random number generators are available for the GPU, e.g., NVIDIA's cuRAND library [16].

A real-time human motion tracking based on depth cameras for data acquisition and PFs for estimation was presented in [1]. Here, the challenge was to estimate a 22D state vector, which is a considerable problem for the PF. To handle this, the authors split the state space into five subspaces and used a PF on each subspace. A comparison between CPU and GPU also showed a speedup of about ten, from 0.5 s processing time per frame down to 0.05 s. Despite these runtime improvements, PFs still suffer from the problem of a constantly large set of particles, which imposes an unavoidable computational burden.

The remainder of this paper is structured as follows. In the next Section, we give a short introduction to the PGF and describe its general work flow. Then, in Sec. III, we describe how to port the PGF to a GPU and what should be taken into account to make the implementation as fast as possible. In Sec. IV, we propose a model to track a sphere and its shape in 3D based on noisy point measurements. This scenario is then used in Sec. V to evaluate the GPU-accelerated PGF against its CPU implementation. Finally, the conclusions are given in Sec. VI.

II. The Progressive Gaussian Filter

In this Section, we can only give a brief introduction to the PGF and its concepts that constitute the particle flow approach. For a more detailed overview of the PGF and its workflow, it is strongly recommended to refer to [13].

The PGF aims to recursively estimate the hidden state $\underline{x}_k$ of a discrete-time stochastic nonlinear dynamic system[3]. In

[1] For example, when using a KF in the nonlinear case the posterior state covariance also depends upon the measurement itself, not only on its corresponding noise covariance.

[2] Again, under the assumption of independent measurement noise.

[3] Here, k denotes the k-th discrete time step and vectors are underlined.

each time step, we receive a set of N_k noisy measurements

$$\mathcal{Y}_k = \{\underline{y}_k^{(1)}, \ldots, \underline{y}_k^{(N_k)}\} \ .$$

Given a prior (i.e., predicted) state estimate

$$f_k^p(\underline{x}_k) := f(\underline{x}_k \,|\, \mathcal{Y}_{k-1}, \ldots, \mathcal{Y}_1) \ ,$$

our goal is to incorporate the measurements $\mathcal{Y}_k$ into the prior to obtain the posterior (i.e., filtered) state estimate

$$f_k^e(\underline{x}_k) := f(\underline{x}_k \,|\, \mathcal{Y}_k, \ldots, \mathcal{Y}_1)$$

by using Bayes' rule

$$f_k^e(\underline{x}_k) \propto f(\mathcal{Y}_k \,|\, \underline{x}_k) \cdot f_k^p(\underline{x}_k) \ . \tag{1}$$

Here, $f(\mathcal{Y}_k \,|\, \underline{x}_k)$ denotes the likelihood function. Unfortunately, solving (1) in closed-form is intractable for arbitrary state densities and likelihood functions. This is the reason why the PGF approximates the predicted as well as the filtered state density as a Gaussian, i.e.,

$$f_k^p(\underline{x}_k) \approx \mathcal{N}(\underline{x}_k; \hat{\underline{x}}_k^p, \mathbf{C}_k^p) \tag{2}$$

and

$$f_k^e(\underline{x}_k) \approx \mathcal{N}(\underline{x}_k; \hat{\underline{x}}_k^e, \mathbf{C}_k^e) \ , \tag{3}$$

with means $\hat{\underline{x}}_k^p$ and $\hat{\underline{x}}_k^e$, and covariance matrices $\mathbf{C}_k^p$ and $\mathbf{C}_k^e$, respectively. Of course, this implies that the PGF, in contrast to Particle Filters, is only capable of maintaining a unimodal state distribution. However, this is also true for the popular (nonlinear) Kalman Filters, and often a unimodal state distribution is sufficient. Also, when linear system models are involved, the PGF allows for a closed-form prediction whereas PFs have to draw random noise samples and propagate each particle individually through the linear model.

Now, how can we compute the posterior Gaussian approximation (3) given the prior Gaussian (2) and the likelihood $f(\mathcal{Y}_k \,|\, \underline{x}_k)$? As closed-form solutions are still not possible even when the prior is Gaussian, a sample-based solution is required. The naïve approach of simply sampling the prior Gaussian, reweighting the samples using the likelihood, and finally approximating the posterior Gaussian using moment matching is not possible as this can also lead to sample degeneration.

The PGF tackles this major problem using the homotopy continuation [17]. Here, we reformulate the Bayes update (1) by introducing the progression parameter $\gamma \in [0, 1]$ according to

$$f_k^e(\underline{x}_k, \gamma) = c(\gamma) \cdot f(\mathcal{Y}_k \,|\, \underline{x}_k)^\gamma \cdot f_k^p(\underline{x}_k) \ ,$$

where $c(\gamma)$ is only a normalization constant. For $\gamma = 0$, we have $f_k^e(\underline{x}_k, 0) = f_k^p(\underline{x}_k)$ which means no information (from the measurements $\mathcal{Y}_k$) is processed, and hence, the posterior equals the prior. In contrast, for $\gamma = 1$ we have $f_k^e(\underline{x}_k, 1) = f_k^e(\underline{x}_k)$, i.e., the unmodified Bayes update, and thus, the entire information from the measurements is fused into the prior state estimate. Furthermore, the above equation can be expressed in terms of a recursive formula

$$f_k^e(\underline{x}_k, \gamma + \Delta) = \frac{c(\gamma + \Delta)}{c(\gamma)} \cdot f(\mathcal{Y}_k \,|\, \underline{x}_k)^\Delta \cdot f_k^e(\underline{x}_k, \gamma) \ , \tag{4}$$

starting with $\gamma = 0$, given step sizes $\Delta > 0$, and $\gamma + \Delta \leq 1$.

Using this, the Bayes update (1) can be approximated by exploiting the fact that the system state is already Gaussian and sampling from such is relatively easy. That is, on the one hand we also approximate any $f_k^e(\underline{x}_k, \gamma)$ as a Gaussian

$$f_k^e(\underline{x}_k, \gamma) \approx \mathcal{N}(\underline{x}_k; \hat{\underline{x}}_k^{(\gamma)}, \mathbf{C}_k^{(\gamma)}) \ . \tag{5}$$

On the other hand, we can compute a Dirac mixture approximation

$$\frac{1}{M} \sum_{i=1}^{M} \delta(\underline{x}_k - \underline{x}_{k,i}^{(\gamma)}) \tag{6}$$

of (5) comprising M equally weighted samples with positions $\underline{x}_{k,i}^{(\gamma)}$. Plugging (6) into (4) yields its Dirac mixture approximation

$$\sum_{i=1}^{M} \frac{c(\gamma + \Delta)}{c(\gamma) M} \cdot f(\mathcal{Y}_k \,|\, \underline{x}_{k,i}^{(\gamma)})^\Delta \cdot \delta(\underline{x}_k - \underline{x}_{k,i}^{(\gamma)}) \ . \tag{7}$$

Computing sample mean and covariance of (7) also yields a Gaussian approximation of $f_k^e(\underline{x}_k, \gamma + \Delta)$.

Now, we can get the desired Gaussian approximation of $f_k^e(\underline{x}_k)$ by starting with prior Gaussian (2), i.e., $f_k^e(\underline{x}_k, 0)$, and recursively computing a Gaussian approximation of (4) until we reach $\gamma = 1$. That is, we perform several resamplings during *one* measurement update[4]. The result is the desired particle flow that moves samples to the important regions of the state space to avoid sample degeneration.

For a better understanding, Algorithm 1 summarizes the PGF filter step procedure. In line 1, the progression starts with the prior mean and covariance. In line 3, the samples $\underline{x}_i$ are computed to approximate the current Gaussian $\mathcal{N}(\hat{\underline{x}}_k^{(\gamma)}, \mathbf{C}_k^{(\gamma)})$. This is done in two steps. First, we compute a set of M equally weighted samples $\underline{s}_i$ approximating a standard normal distribution of appropriate dimension offline (before filter execution) using the LCD approach [18], which also serves as basis for the S^{2}KF [8]. Second, the samples $\underline{s}_i$ are transformed online (during filter execution) to the samples $\underline{x}_i$ with the aid of the Mahalanobis transformation [19]. The evaluation of the log-likelihood and the determination of the minimum and maximum log-likelihood value that are required to obtain the step size Δ are performed in lines 4 to 9. When the extremes are equal or all samples are evaluated to $-\infty$ (i.e., a likelihood value of zero), there is no particle flow direction, and hence, no progression is possible. The progression increment Δ for the current iteration is computed in lines 10 to 14. After that, in lines 15 to 19, the new sample weights α_i are computed, and used to obtain the next Gaussian approximation $\mathcal{N}(\hat{\underline{x}}_k^{(\gamma+\Delta)}, \mathbf{C}_k^{(\gamma+\Delta)})$ by means of moment matching. The progression stops when γ reaches 1. The result is the updated Gaussian state estimate $f_k^e(\underline{x}_k)$.

III. Progressive Gaussian Filtering on the GPU

In Sec. II, we described the idea and the algorithm of the PGF. In this Section, we focus on how to offload the PGF onto a GPU in order to speed up the state estimator and reduce the workload of the CPU. Here, we only deal with the PGF's measurement update. Of course, the state prediction can be performed on the GPU, too. However, as we need to copy the

[4]This can be compared with Particle Filters, which perform only one resampling after an update (if any).

filtered state estimate back to the CPU to save and/or process it anyway, we can also perform the prediction on the CPU and copy the predicted state estimate back to the GPU for the next measurement update. This holds in particular when the prediction is easy to compute, e.g., in case of a linear system model.

When writing code for a GPU, some aspects have to be considered.

1) Avoid copying data between CPU and GPU memory whenever possible. The PCI express bus is very slow compared to the usual memory access, and it can take milliseconds to transfer data.
2) To best profit from its massive parallel execution capabilities, keep the GPU busy with arithmetic operations to hide memory access latencies by running hundreds or thousands of threads at a time.
3) Try to reduce the GPU memory access. Performing computations multiple times can be better instead of sharing results between threads.
4) Ensure that threads follow the same control flow whenever possible. Threads are executed in groups, and threads of a group are stalled when the control flow of the group diverges, e.g., when not all threads of a group take the same branches.

In the following, we go step by step through Algorithm 1 and discuss porting it to the GPU. Not the entire procedure will be executed on the GPU. All lines marked as orange remain on the CPU.

Before entering the while loop, we transfer the predicted state mean $\hat{\underline{x}}_k^p$ and covariance matrix $\mathbf{C}_k^p$, the pre-computed standard normal samples $\underline{s}_i$ (column-wise stored in single matrix $\mathbf{V}$), the measurements $\mathcal{Y}_k$, and maybe other likelihood related data such as noise covariances from the CPU to the GPU. In particular, if the number of samples M does not change over time, the samples $\underline{s}_i$ have to be transferred only once for the entire program execution (as will be case in our evaluation in Sec. V). The while loop itself is executed on the CPU and enqueues all the necessary function calls on the GPU.

First, the Cholesky decomposition of $\mathbf{C}$ has to be computed. Unfortunately, this is a recursive procedure, and hence, only the data can be read and written in parallel. The Cholesky decomposition itself is computed by a single thread. Then, each sample $\underline{x}_i$ can be computed using a parallized matrix-matrix multiplication $\sqrt{\mathbf{C}} \cdot \mathbf{V}$ followed by a parallized column-wise addition of the mean vector $\hat{\underline{x}}$. Hence, the samples $\underline{x}_i$ are likewise stored column-wise in a matrix $\mathbf{M}$.

Also, the log-likelihood evaluation (line 4) can be basically performed in parallel. However, this is problem-specific and can lead to different implementations for different log-likelihoods. We will discuss the implementation of the log-likelihood of our evaluation in Sec. V. For finding the minimum and maximum log-likelihood value, we exploit the usual parallel reduction scheme that requires only a logarithmic number of operations, e.g., [14].

The log-likelihood extremes ($l_{\min}$ and $l_{\max}$) are then copied to the CPU, and the step size Δ of the current progression step is computed. This is done as we need Δ on the CPU anyway to increment γ in order to determine end of the progression. The

Algorithm 1 Progressive Gaussian Filter

1: Set $\hat{\underline{x}} = \hat{\underline{x}}_k^p, \mathbf{C} = \mathbf{C}_k^p, \gamma = 0$
2: **while** $\gamma < 1$ **do**
3: $\quad \underline{x}_i = \sqrt{\mathbf{C}} \cdot \underline{s}_i + \hat{\underline{x}} \quad \forall\, 1 \leq i \leq M$
4: $\quad l_i = \log(f(\mathcal{Y}_k \,|\, \underline{x}_i)) \quad \forall\, 1 \leq i \leq M$
5: $\quad \mathcal{S} = \{l_i \mid \forall\, 1 \leq i \leq M \ \wedge\ l_i > -\infty\}$
6: $\quad l_{\min} = \min(\mathcal{S}) \quad l_{\max} = \max(\mathcal{S})$
7: $\quad$ **if** $\mathcal{S} = \emptyset \ \vee\ l_{\min} = l_{\max}$ **then**
8: $\quad\quad$ No progression possible $\Rightarrow$ Abort update.
9: $\quad$ **end if**
10: $\quad \Delta = -\log(M) \ / \ (l_{\min} - l_{\max})$
11: $\quad$ **if** $\gamma + \Delta > 1$ **then**
12: $\quad\quad \Delta = 1 - \gamma$
13: $\quad$ **end if**
14: $\quad \gamma = \gamma + \Delta$
15: $\quad f(\mathcal{Y}_k \,|\, \underline{x}_i)^\Delta = \exp((l_i - l_{\max}) \cdot \Delta) \quad \forall\, 1 \leq i \leq M$
16: $\quad \alpha = \sum_{j=1}^{M} f(\mathcal{Y}_k \,|\, \underline{x}_j)^\Delta$
17: $\quad \alpha_i = f(\mathcal{Y}_k \,|\, \underline{x}_i)^\Delta \ / \ \alpha \quad \forall\, 1 \leq i \leq M$
18: $\quad \hat{\underline{x}} = \sum_{i=1}^{M} \alpha_i \cdot \underline{x}_i$
19: $\quad \mathbf{C} = \sum_{i=1}^{M} \alpha_i \cdot (\underline{x}_i - \hat{\underline{x}}) \cdot (\underline{x}_i - \hat{\underline{x}})^\top$
20: **end while**
21: Set $\hat{\underline{x}}_k^e = \hat{\underline{x}}, \mathbf{C}_k^e = \mathbf{C}$

step size Δ is then transferred back to the GPU and used to compute $f(\mathcal{Y}_k \,|\, \underline{x}_i)^\Delta$ for all samples in parallel. The required sum of sample weights α is again computed by means of the paralell reduction scheme. The subsequent sample weight normalization can be computed in parallel for each sample, and the resulting weights α_i are stored in the row vector $\underline{\alpha}$.

The new sample mean can now be obtained with the a parallel matrix-vector multiplication $\hat{\underline{x}} = \mathbf{M} \cdot \underline{\alpha}^\top$. A parallel column-wise subtraction of $\mathbf{M}$ with $\hat{\underline{x}}$ is stored in a matrix $\mathbf{D}$, and yields the differences $\underline{x}_i - \hat{\underline{x}}$. A subsequent row-wise parallel multiplication of $\underline{\alpha}$ with $\mathbf{D}$ is stored in $\mathbf{D}_\alpha$. The parallelized matrix-matrix multiplication $\mathbf{C} = \mathbf{D}_\alpha \cdot \mathbf{D}^\top$ yields the new sample covariance, and the next progression step begins.

Finally, after the while loop is completed, we transfer the filtered state mean $\hat{\underline{x}}_k^e$ and covariance matrix $\mathbf{C}_k^e$ back to the CPU. In summary, all the expensive computations are moved to the GPU, and the data transfer between CPU and GPU during the progression is reduced to an exchange of a few scalars, namely $l_{\min}$, $l_{\max}$, and Δ.

IV. Extended Object Tracking

In order to properly evaluate the GPU version of the PGF, our goal is to do extended object tracking in real-time when processing a large number of measurements. Here, we try to estimate the position $\underline{c}_k = [c_k^x, c_k^y, c_k^z]^\top$, the radius r_k, and the velocity $\underline{\nu}_k = [\nu_k^x, \nu_k^y, \nu_k^z]^\top$ of a sphere in 3D (see Figures 1 and 2). Hence, the system state vector is given by

$$\underline{x}_k = [\underline{c}_k^\top, r_k, \underline{\nu}_k^\top]^\top \ .$$

In each time step, we receive a set of noisy point measurements $\mathcal{Y}_k$ in x-y-z-coordinates from one ore more depth

cameras with known locations. That is, each measurement $\underline{y}_k^{(i)}$ was observed by one of these cameras.

We assume that each single measurement $\underline{y}_k^{(i)}$ stems from a source $\underline{z}_k^{(i)}$ on the sphere's surface and is corrupted by additive zero-mean Gaussian white noise $\underline{v}_k^{(i)}$ with covariance matrix $\mathbf{R}_k^{(i)} \in \mathbb{R}^{3\times3}$ according to

$$\underline{y}_k^{(i)} = \underline{z}_k^{(i)} + \underline{v}_k^{(i)} \ .$$

For different measurements $\underline{y}_k^{(i)}$, it is assumed that their measurement noises $\underline{v}_k^{(i)}$ are mutually independent. Consequently, the likelihood becomes a product of N_k independent likelihoods (one for each measurement)

$$f(\mathcal{Y}_k \,|\, \underline{x}_k) = \prod_{i=1}^{N_k} \mathcal{N}(\underline{y}_k^{(i)} - \underline{z}_k^{(i)}; \mathbf{R}_k^{(i)}) \ . \tag{8}$$

The corresponding log-likelihood required by the PGF can be evaluated in closed-form to

$$\begin{aligned} \log(f(\mathcal{Y}_k \,|\, \underline{x}_k)) = \sum_{i=1}^{N_k} & -(\frac{3}{2}\log(2\pi) + \frac{1}{2}\log(|\mathbf{R}_k^{(i)}|)) \\ & - \frac{1}{2}(\underline{y}_k^{(i)} - \underline{z}_k^{(i)})^\top (\mathbf{R}_k^{(i)})^{-1} (\underline{y}_k^{(i)} - \underline{z}_k^{(i)}) \ . \end{aligned} \tag{9}$$

It is important to note that this likelihood penalizes state estimates $\underline{x}_k$ where measurements $\underline{y}_k^{(i)}$ and their associated sources $\underline{z}_k^{(i)}$ are not close together (i.e., when they have a large Mahalanobis distance). Also note that the inverse of $\mathbf{R}_k^{(i)}$ and its determinant $|\mathbf{R}_k^{(i)}|$ can be computed analytically.

Up to now, we have assumed that we know the source $\underline{z}_k^{(i)}$ of each measurement $\underline{y}_k^{(i)}$. Unfortunately, this is not true as we do not use any type of unique markers allowing for a correct association. We only have an unlabeled point cloud of the sphere. However, in order to get the most probable source for each measurement, we make use of the so-called Greedy Association Model (GAM) [20]. To approximate the true (but unknown) source, we do not only rely on the sphere estimate $\underline{x}_k$ but also on the measurement itself (hence the term greedy). Additionally, we make also use of the known camera location $\underline{p}$ to exploit the geometrical interaction between camera, measurement, and sphere according to

$$\underline{z}_k^{(i)} \approx \text{computeSource}(\underline{x}_k, \underline{p}, \underline{y}_k^{(i)}) \ .$$

Here, we roughly assume that the camera can only observe points from one half of the sphere, namely the half between the camera and the "visibility plane" which is orthogonal to the direction from camera location $\underline{p}$ to sphere center $\underline{c}_k$ (see Fig. 2). So only the points on this side of the sphere can be a possible source for a measurement. To select the most probable source $\underline{z}_k^{(i)}$ on the determined sphere side for the measurement $\underline{y}_k^{(i)}$, we have to distinguish between four possible cases (a) – (d) depicted in Fig. 2:

(a) its projection line intersects the sphere, and the measurement lies between camera and sphere,
(b) its projection line intersects the sphere, and the sphere lies between camera and measurement,

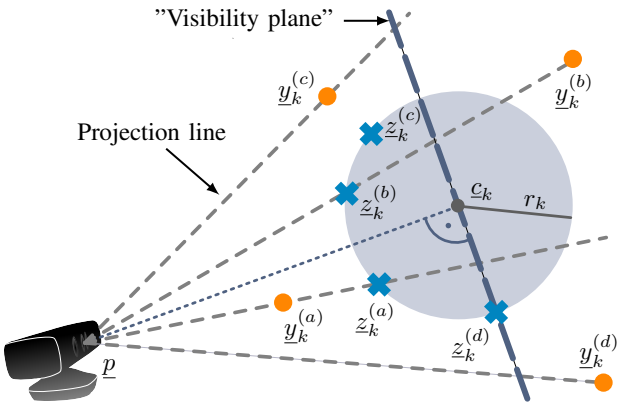

Figure 2: The used GAM with sphere center $\underline{c}_k$ and radius r_k, camera position $\underline{p}$, received measurements $\underline{y}_k^{(i)}$ (orange circles), and their associated sources $\underline{z}_k^{(i)}$ (blue crosses).

(c) its projection line has no intersection with the sphere, and the measurement lies in front of the "visibility plane", and finally
(d) its projection line has no intersection with the sphere, and the measurement lies behind the "visibility plane".

Algorithm 2 shows the source computation in detail, which has to be individually performed for each received measurement $\underline{y}_k^{(i)}$ and system state $\underline{x}_k$. The proposed approach does not simply associate the nearest point on the sphere to a measurement regardless if this point is even visible by the camera or not. More precisely, it penalizes any estimate $\underline{x}_k$ where measurements would lie behind the estimated sphere, i.e., are physically impossible to measure, by choosing a source that is far away from the measurement, e.g., case (b).

So on the one hand, the proposed log-likelihood is rather complicated and, paired with many measurements, its evaluation requires a substantial amount of computational resources. But on the other hand, its usage should result in a good estimation performance due to its detailed modelling of the physical background.

V. Evaluation

We want to compare the GPU implementation of the PGF with its CPU implementation by means of tracking a sphere and its shape in 3D based on the model presented in Sec. IV. For that reason, we simulate a sphere's movement along a nonlinear path observed by five virtual depth cameras that are modeled to behave like Microsoft Kinect cameras (see Fig. 1). Due to the high resolution of a Kinect camera (640×480 pixels) and the possibility that the sphere can be seen from several cameras at a time, it is likely that we receive a large number of measurements per time step, making this scenario a demanding estimation problem.

A. System Model

We describe the temporal evolution of the sphere using a nearly constant velocity model

$$\underline{x}_k = \mathbf{A}\underline{x}_{k-1} + \mathbf{B}\underline{w} \ ,$$

with matrices

$$\mathbf{A} = \begin{bmatrix} \mathbf{I}_{3\times3} & 0 & T\cdot\mathbf{I}_{3\times3} \\ \mathbf{0}_{1\times3} & 1 & \mathbf{0}_{1\times3} \\ \mathbf{0}_{3\times3} & \mathbf{0}_{3\times1} & \mathbf{I}_{3\times3} \end{bmatrix}$$

and

$$\mathbf{B} = \begin{bmatrix} T\cdot\mathbf{I}_{3\times3} & \mathbf{0}_{3\times1} \\ \mathbf{0}_{1\times3} & 1 \\ \mathbf{I}_{3\times3} & \mathbf{0}_{3\times1} \end{bmatrix} ,$$

and time-invariant, zero-mean, and white Gaussian noise for velocity and radius $\underline{w} = [w^x, w^y, w^z, w^r]^\top$ with covariance

$$\mathbf{Q} = \mathrm{diag}(1, 1, 1, 10^{-5}) .$$

The radius noise w^r gives the state estimator the ability to detect changes in the sphere's extent over time. Here, the time period T is set to 0.03 s which means we get a new set of measurements approx. 33 times per second. As the system model is linear and corrupted by Gaussian noise, and the PGF maintains only a Gaussian distributed system state, we can compute the time update optimally in closed-form.

B. Measurement Model

To incorporate measurements into the state estimate, we utilize the likelihood function (8). The measurement noise covariance matrices $\mathbf{R}_k^{(i)}$ are obtained according to the Kinect sensor model proposed in [21]. The result is a more demanding but realistic non-isotropic measurement noise allowing for an adequate simulation.

C. GPU Implementation Details

In Sec. III, we already discussed the GPU implementation of the PGF itself. What is still missing is an efficient implementation of the required log-likelihood (9). First of all, the inverse and determinant of the N_k measurement noise covariances $\mathbf{R}_k^{(i)}$ are required in order to evaluate the log-likelihood for a given set of measurements. This can be perfectly done in parallel for each noise covariance on the GPU, resulting in a reduced runtime of the measurement update.

Generally, the log-likelihood can be evaluated completely independent for each state sample $\underline{x}_i$. However, compared to PFs with its many particles the PGF uses only a very small number of samples M per progression step. Additionally, we focus on situations in which the number of measurements N_k is very large, i.e., in the order of several thousands. When now simply evaluating the log-likelihood (9) for each of the M state samples $\underline{x}_i$ in parallel, we could only schedule a very small number of threads on the GPU. Moreover, each thread has to compute the large sum of N_k terms. Hence, the GPU cannot fully exploit its resources, and thus, this approach is inadvisable. We tackle this problem by evaluating each part of the log-likelihood sum in parallel. That is, we schedule N_k threads where each thread evaluates one term of the log-likelihood sum M times (once for each state sample $\underline{x}_i$). The results of all threads are collected in a large matrix $\mathbf{E}$ of dimension $N_k \times M$. Subsequently, we perform a parallel sum reduction along each column of $\mathbf{E}$ to obtain the final log-likelihood evaluation for each state sample $\underline{x}_i$. Although this approach involves more memory accesses (the entries of $\mathbf{E}$ first have to be written to the memory and subsequently read step by step to perform the reduction), it requires only $M \cdot \log(N_k)$ sum operations instead of $M \cdot N_k$. Combined with the much better utilization of the GPU (consider the thousands of threads operating in parallel) it is much faster compared to the naïve approach where only M threads would be executed in parallel.

Algorithm 2 computeSource

Input: State $\underline{x}_k$, camera position $\underline{p}$, and measurement $\underline{y}_k^{(i)}$

```
d = normalize(y_k^(i) − p)   e = y_k^(i) − c_k
// Check for line-sphere-intersection
a = d^⊤ · e   b = a^2 − (e^⊤ · e − r_k^2)
if b < 0 then
    // No line-sphere-intersection
    n = normalize(c_k − p)        // "Visibility plane" normal
    if e^⊤ · n < 0 then
        // In front of "visibility plane"
        // Project measurement on sphere
        z_k^(i) = c_k + r_k · normalize(e)        // Case (c) in Fig. 2
    else
        // Behind "visibility plane"
        // Intersection of projection line and "visibility plane"
        l = y_k^(i) − (e^⊤ · n / d^⊤ · n) · d
        // Project intersection on sphere
        z_k^(i) = c_k + r_k · normalize(l − c_k) // Case (d) in Fig. 2
    end if
else
    // Line-sphere-intersection
    d_1 = −a + √b   d_2 = −a − √b
    // Choose intersection closest to the camera position
    if d_1 < d_2 then
        z_k^(i) = y_k^(i) + d_1 · d                // Case (a) in Fig. 2
    else
        z_k^(i) = y_k^(i) + d_2 · d                // Case (b) in Fig. 2
    end if
end if
```

Output: Measurement source $\underline{z}_k^{(i)}$

D. Setup

The tracking setup is as follows. The sphere has an initial radius of 15 cm and moves, in total, for 5 s along a nonlinear path observed by the five Kinect cameras as shown in Fig. 1. After 2 s, the sphere radius increases over the next 0.75 s from 15 cm to 40 cm. At 4 s, the sphere radius shrinks abruptly back to 15 cm. This allows for more changes in the number of available measurements and to investigate how well the PGF implementations can handle such shape changes.

An initial state estimate is computed using the first set of available measurements. That is, the position mean and covariance are set to the sample mean and sample covariance of the measurements, the radius mean and variance is obtained by computing sample mean and variance of the norm between the initial position and each measurement. The initial velocity mean is set to zero and its covariance matrix to the identity. Moreover, position, radius, and velocity are initially uncorrelated.

The CPU implementation of the PGF in C++ is based on the Eigen linear algebra library [22] that makes heavy use of

SIMD instructions via SSE. Additionally, the evaluation of the log-likelihood is effectively parallized using OpenMP. The evaluation is performed on an Intel Core i7-3770 (3.4 GHz, 4 cores, 8 threads).

The GPU implementation is based on OpenCL 1.2 and is executed on an AMD Radeon R9 280X graphics card, a mid-class GPU for about $250 that, however, offers a great double-precision performance compared to other GPUs. The linear state prediction is executed on the CPU.

Both the CPU and the GPU implementation use double-precision arithmetics and the same set of $M = 51$ samples $\underline{s}_i$ approximating the standard normal distribution. We perform 50 Monte Carlo runs.

E. Results

To get fair results, the GPU runtimes comprise the entire state estimation procedure, not only the parts executed on the GPU, that is,

1) the state prediction on the CPU,
2) the time needed to transfer the predicted state estimate as well as the measurements and their associated camera positions and noise covariance matrices to the GPU,
3) the measurement update itself, and
4) the time needed to copy the filtered state estimate back to the CPU.

The minimum runtime (lower bound of a curve) and maximum runtime (upper bound of a curve) of the PGF implementations are depicted in Fig. 3. It can be seen that the GPU implementation has a much lower runtime than the CPU implementation. The GPU variant required 110 ms at most, whereas the CPU required up to 1.6 s. Moreover, the GPU has a much smaller runtime variance, especially for a larger number of measurements. To be real-time capable in this scenario, a maximum runtime of 30 ms is allowed. Here, the CPU is often far away from that. More precisely, the CPU is only 8% of the time below the real-time limit. On the contrary, the GPU achieves this 68% of the time. Although the GPU variant has to exchange data with the CPU, the overall minimum runtime is only 5 ms compared to the CPU with a minimum runtime of 16 ms.

Fig. 3 also shows the number of the processed measurements over time. The number of measurements ranges from 1,000 up to 42,000. More precisely, when the sphere radius increases after 2 s, we receive an increasing number of measurements as the sphere's surface gets larger and larger. Note also the radical drop at 4 s when the sphere shrinks back to its initial radius. Due to the varying distance to each camera, the number of measurements changes even when the size of the sphere is constant, e.g., between 0 s and 2 s, or 3 s to 4 s. It can be seen that the CPU runtime and the number of processed measurements is highly correlated. This is due to the fact that 1k measurements already keep all cores of the CPU busy. The GPU, however, is rather unaffected by those changes, as it has still many computational resources. Keep also in mind that processing 42k measurements at once with a KF is intractable as the measurement covariance matrix would require several gigabytes of data, not to mention the time to process it.

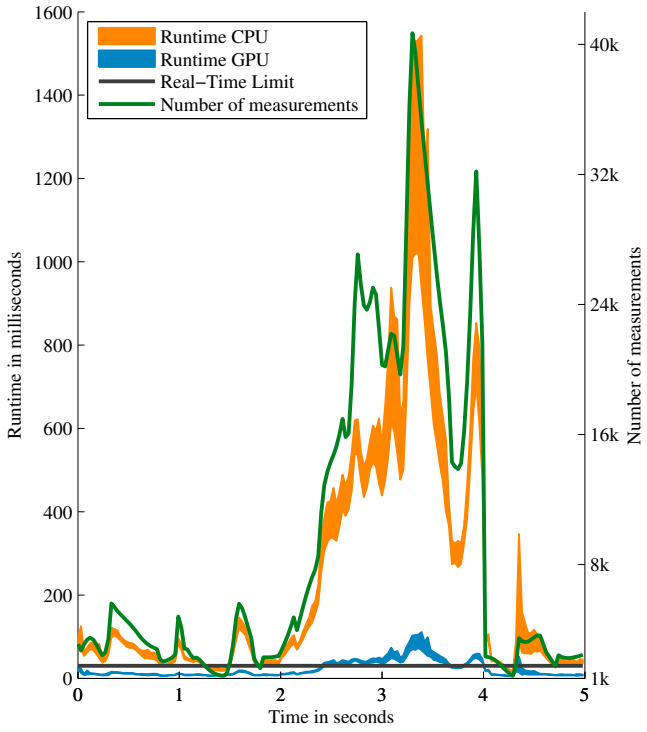

Figure 3: Minimum and maximum PGF runtimes as well as the number of processed measurements.

Fig. 4 depicts the speedup of the PGF over the number of processed measurements. The speedup shows a logarithmic characteristic and reaches already an average value of 10 when processing 10k measurements. The GPU is always faster than the CPU and speedups over 20 are possible. Hence, the PGF delivers the same speedups as for the Particle Filters, if not better.

The estimation accuracy of the PGF implementations is also of interest. For position and radius the Root Mean Square Errors (RMSEs) are shown in Fig. 5. First of all, the errors in general are very small due to the large number of measurements. Only the sphere's abrupt shrinking at 4 s causes a significant jump in the errors. Moreover, one can see that both variants, CPU and GPU, offer nearly identical estimation errors (besides roundoff errors due to the different order of the arithmetic operations). This is expected, and hence, both implementations are assumed to work properly. Thus, regarding numerical issues, it should be no problem to switch to the GPU-accelerated variant of the PGF in other applications.

VI. Conclusions

In this paper, we presented a powerful GPU implementation of the PGF allowing for real-time extended object tracking. First, we motivated the use of nonlinear state estimators which rely on likelihood functions to process multiple measurements in a single filter step in order to increase the estimation performance. After that, we briefly described the PGF with its particle flow approach as a serious alternative to the classical Particle Filters.

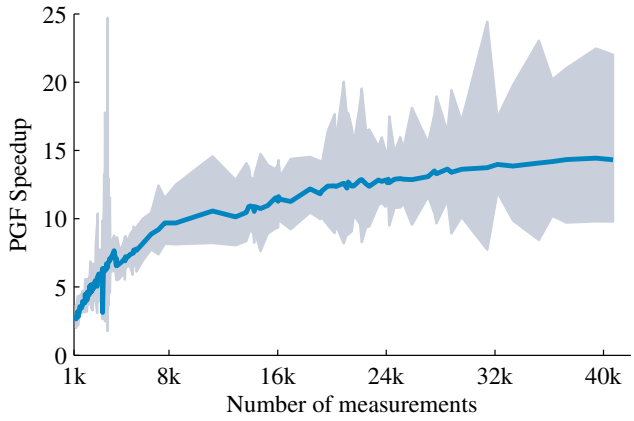

Figure 4: Minimum, maximum, and average speedup of the GPU-accelerated PGF.

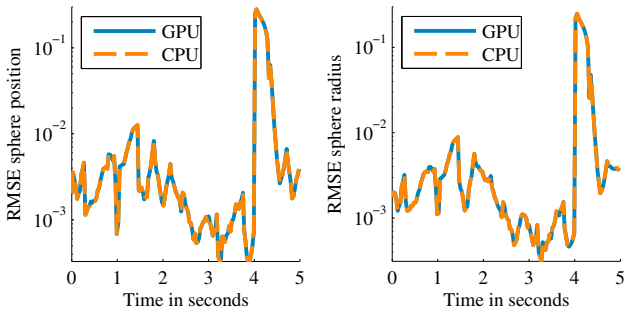

Figure 5: Sphere estimation errors.

Like other estimators, also the PGF can be boosted by executing it on a GPU, so we gave an overview of how to port it effectively to a GPU. Moreover, we proposed a likelihood to estimate pose and extend of a sphere in 3D based on noisy point measurements. This likelihood was then used to evaluate the GPU variant of the PGF in a real-world tracking scenario. The evaluation showed that the CPU implementation is far from being real-time capable, whereas the GPU implementation almost is, even when processing tens of thousands of measurements at a time.

Of course, the PGF and its GPU-accelerated variant are not limited to the application of extended object tracking. For example, they can be used to get a maximum a posteriori (MAP) estimate by omitting any prediction and executing only one measurement update but with maybe millions of measurements at once (i.e, batch processing). This can be a way to replace the classical maximum likelihood (ML) approach to do parameter estimation with the advantages of providing prior information and getting the uncertainty of the parameter estimate.

References

[1] Licong Zhang, Jürgen Sturm, Daniel Cremers, and Dongheui Lee, "Real-Time Human Motion Tracking Using Multiple Depth Cameras," in *Proceedings of the 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2012)*, Vilamoura, Portugal, Oct. 2012, pp. 2389–2395.

[2] Marcus Baum and Uwe D. Hanebeck, "Extended Object Tracking with Random Hypersurface Models," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 50, no. 1, pp. 149–159, Jan. 2014.

[3] Michael Feldmann, Dietrich Fränken, and Wolfgang Koch, "Tracking of Extended Objects and Group Targets Using Random Matrices," *IEEE Transactions on Signal Processing*, vol. 59, no. 4, pp. 1409–1420, Apr. 2011.

[4] K. Gilholm and D. Salmond, "Spatial Distribution Model for Tracking Extended Objects," *IEE Proceedings Radar, Sonar and Navigation*, vol. 152, no. 5, pp. 364–371, Oct. 2005.

[5] Yaakov Bar-Shalom, X. Rong Li, and Thiagalingam Kirubarajan, *Estimation with Applications to Tracking and Navigation*. New York Chichester Weinheim Brisbane Singapore Toronto: Wiley-Interscience, 2001.

[6] Dan Simon, *Optimal State Estimation*, 1st ed. Wiley & Sons, 2006.

[7] Simon J. Julier and Jeffrey K. Uhlmann, "Unscented Filtering and Nonlinear Estimation," in *Proceedings of the IEEE*, vol. 92, Mar. 2004, pp. 401–422.

[8] Jannik Steinbring and Uwe D. Hanebeck, "LRKF Revisited: The Smart Sampling Kalman Filter (S^2KF)," *Journal of Advances in Information Fusion*, vol. 9, no. 2, pp. 106–123, Dec. 2014.

[9] Branko Ristic, Sanjeev Arulampalam, and Neil Gordon, *Beyond the Kalman Filter: Particle Filters for Tracking Applications*. Artech House Publishers, 2004.

[10] Arnaud Doucet and Adam M. Johansen, "A Tutorial on Particle Filtering and Smoothing: Fifteen Years Later," in *Oxford Handbook of Nonlinear Filtering*, 2011, pp. 656–704.

[11] Jayesh H. Kotecha and Petar M. Djuric, "Gaussian Particle Filtering," *IEEE Transactions on Signal Processing*, vol. 51, no. 10, pp. 2592–2601, Oct. 2003.

[12] Uwe D. Hanebeck, "PGF 42: Progressive Gaussian Filtering with a Twist," in *Proceedings of the 16th International Conference on Information Fusion (Fusion 2013)*, Istanbul, Turkey, Jul. 2013.

[13] Jannik Steinbring and Uwe D. Hanebeck, "Progressive Gaussian Filtering Using Explicit Likelihoods," in *Proceedings of the 17th International Conference on Information Fusion (Fusion 2014)*, Salamanca, Spain, Jul. 2014.

[14] Gustaf Hendeby, Rickard Karlsson, and Fredrik Gustafsson, "Particle Filtering: The Need for Speed," *EURASIP Journal on Advances in Signal Processing*, vol. 2010, Feb. 2010.

[15] Matthew A. Goodrum, Michael J. Trotter, Alla Aksel, Scott T. Acton, and Kevin Skadron, "Parallelization of Particle Filter Algorithms," in *Computer Architecture*, ser. Lecture Notes in Computer Science Volume 6161, 2012, pp. 139–149.

[16] NVIDIA Corporation, "cuRAND library," Mar. 2015. [Online]. Available: http://docs.nvidia.com/cuda/curand/

[17] Jorge Nocedal and Stephen J. Wright, *Numerical Optimization*, 2nd ed., ser. Springer Series in Operations Research and Financial Engineering. Springer, 2006.

[18] Uwe D. Hanebeck, Marco F. Huber, and Vesa Klumpp, "Dirac Mixture Approximation of Multivariate Gaussian Densities," in *Proceedings of the 2009 IEEE Conference on Decision and Control (CDC 2009)*, Shanghai, China, Dec. 2009.

[19] Wolfgang Härdle and Léopold Simar, *Applied Multivariate Statistical Analysis*, 2nd ed. Berlin Heidelberg: Springer, 2008.

[20] Florian Faion, Antonio Zea, and Uwe D. Hanebeck, "Reducing Bias in Bayesian Shape Estimation," in *Proceedings of the 17th International Conference on Information Fusion (Fusion 2014)*, Salamanca, Spain, Jul. 2014.

[21] Florian Faion, Simon Friedberger, Antonio Zea, and Uwe D. Hanebeck, "Intelligent Sensor-Scheduling for Multi-Kinect-Tracking," in *Proceedings of the 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2012)*, Vilamoura, Algarve, Portugal, Oct. 2012, pp. 3993–3999.

[22] "Eigen C++ linear algebra library," Mar. 2015. [Online]. Available: http://eigen.tuxfamily.org/