

Hardware Acceleration of Monte-Carlo Sampling for Energy Efficient Robust Robot Manipulation

Yanqi Liu
Dept. of Computer Science
Brown University
Providence, RI, USA
yanqi_liu@brown.edu

Giuseppe Calderoni
Dept. of Automation and Informatics
Politecnico di Torino
Torino, Italy
giuseppe.calderoni@studenti.polito.it

R. Iris Bahar
School of Engineering
Dept. of Computer Science
Brown University
Providence, RI USA
iris_bahar@brown.edu

Abstract—Algorithms based on Monte-Carlo sampling have been widely adapted in robotics and other areas of engineering due to their performance robustness. However, these sampling-based approaches have high computational requirements, making them unsuitable for real-time applications with tight energy constraints. In this paper, we investigate 6 degree-of-freedom (6DoF) pose estimation for robot manipulation using this method, which uses rendering combined with sequential Monte-Carlo sampling. While potentially very accurate, the significant computational complexity of the algorithm makes it less attractive for mobile robots, where runtime and energy consumption are tightly constrained. To address these challenges, we develop a novel hardware implementation of Monte-Carlo sampling on an FPGA with lower computational complexity and memory usage, while achieving high parallelism and modularization. Our results show 12X–21X improvements in energy efficiency over low-power and high-end GPU implementations, respectively. Moreover, we achieve real time performance without compromising accuracy.

Index Terms—Robotics, Monte-Carlo sampling, Low-power

I. INTRODUCTION

Robot manipulation tasks generally involve three stages: object recognition, pose estimation, and object manipulation. Convolutional Neural Networks (CNNs) have shown high accuracy and fast inference speed in object recognition, making their use widely popular for robotic applications, such as picking up and manipulating objects. However, CNNs also have several shortcomings, including extensive training effort, opacity in decision making and inability to recover from incorrect decisions. Moreover, CNNs tend to overfit to the training data due to their high non-linearity and parameter counts [1]. Overfitting also makes the CNN vulnerable to adversarial attack (e.g., via small image perturbations [2], [3]), and can also lead to poor predictions when faced with unfamiliar scenarios. In particular, when the robot operates in the real world, it is subject to complex and changing environments that often have not been captured by training data.

Alternatively, discriminative-generative algorithms [4], [5], [6] offer a promising solution to achieve robust performance. Such methods combine the discriminative power of inference (using deep neural networks) with generative Monte-Carlo sampling to achieve robust and adaptive perception.

This work is supported by equipment grants from Nvidia and Xilinx Corporations, and a grant through the Brown University Office of Research Development.

In particular, the Monte-Carlo sampling stage can recover from the false negatives obtained from neural network outputs and offers an explainable final decision. For instance, the discriminative-generative approach of [6] demonstrated over a 50% improvement in pose estimation accuracy compared to end-to-end neural network approaches, which enables robust robot manipulation under various environmental changes. However, while neural network inference can be completed within a second on modern general purpose graphic processing units (GPUs), the iterative process of Monte-Carlo sampling does not map well to GPU acceleration, making the algorithm less amenable to meeting the energy and real-time constraints required of mobile applications. In particular, the run time and energy consumption is determined by the range of sampling, the number of iterations, and the computational complexity of the likelihood function. Instead, some other means of hardware acceleration is required to make Monte-Carlo sampling fast as well as energy efficient.

Custom hardware implementations (using FPGAs or ASICs) can operate with reduced energy consumption, even while running at a lower clock frequency, since they have better dataflow flexibility than GPUs or CPUs. However, a direct translation from the software implementation to hardware often is hardly able to yield any improvements. This paper describes a novel FPGA implementation of Monte-Carlo sampling that provides the same accuracy as GPU-CPU approaches such as [7], but with significantly improved runtime and energy consumption. This paper makes the following contributions:

- We develop a complete Monte-Carlo generative inference flow suitable for hardware acceleration on an FPGA.
- We demonstrate how pipelining, numerical quantization, partial rasterization, and image storage optimizations can be used to significantly reduce computational complexity and memory utilization of the generative algorithm.
- We show how to partition the algorithm using multiple parallel customized processing cores to increase throughput and memory access efficiency.
- We show that our FPGA Monte-Carlo sampling design achieves a 12X–21X improvement in energy efficiency compared to GPU-CPU implementations, while providing real time performance with no accuracy loss.

II. BACKGROUND

Robot perception is an important step for robot manipulation in unstructured environments. In particular, object pose estimation is the key step for robot manipulation. Learning-based methods have been used based on end-to-end neural networks. For instance, PoseCNN [8] proposes an network that learns the object segmentation with 3D translation and rotation. DOPE [9] focuses on performance in dark environments by training on synthetic data from domain randomization and photo-realistic simulation. DenseFusion [10] concatenates features extracted from object segmentation and point clouds to estimate the pose from this hybrid RGB-depth representation of the object. While the end-to-end network based methods can achieve real time performance on GPUs, they require relatively large training sets for 6 DoF object poses. Moreover, the network accuracy may be severely affected under challenging natural environments (e.g. change of lighting conditions and objects occlusion) as evaluated in [6].

In this paper, we focus on discriminative-generative methods, where neural network output is followed with a probabilistic inference in a two-stage paradigm. This approach is fundamentally different from the end-to-end learning network approaches proposed in [8], [9], [10], where the performance largely depends on network accuracy, so there is no way to recover once it has made a false decision. Techniques based on discriminative-generative methods include the work of [7], which proposed to use a pyramidCNN to generate a probability heatmap of the object, followed by a bootstrap filter to find the optimal object pose from the object distribution. GRIP [6] further improves the performance of [7] in dark, occluded scenes by exploiting point cloud features.

Pose estimation is an important step for real-time systems, yet there is little work that considers how it may be accelerated in hardware, and these approaches are either not accurate enough for such tasks as robot manipulation [11], provide only partial solutions (e.g., [12], [13]), or cannot be integrated with a discriminative-generate approach [14], which is especially useful for reasoning in unstructured environments.

A generative Monte-Carlo approach provides a greater search space and explainable reasoning, which improves accuracy and robustness, but at the expense of computational complexity. Particle filtering (an application of Monte-Carlo sampling) has been implemented on FPGAs for accelerating object tracking and robot mapping and localization [15], [16], [17], though not for pose estimation. Our goal is to develop a novel FPGA design for 6 DoF object pose estimation based on Monte-Carlo sampling that achieves real time performance with significantly reduced energy consumption.

III. ALGORITHM

The two-stage paradigm for the discriminative-generative algorithm proposed in [7] is shown in Fig. 1. Our goal is to design an efficient hardware implementation of Monte-Carlo sampling used in the second stage of the algorithm for 6 DoF object pose estimation. The input to this Monte-Carlo generative sampling algorithm is a series of bounding boxes

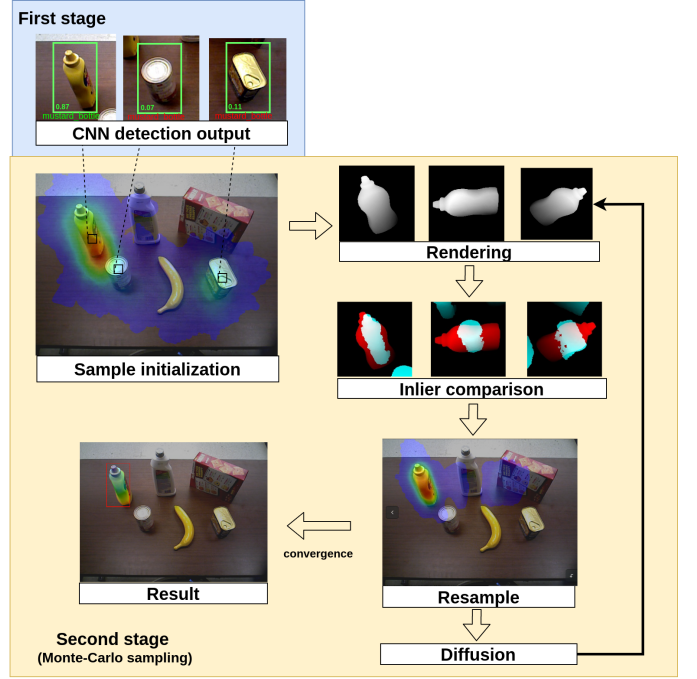


Fig. 1: Two-stage paradigm. 1st stage uses CNN for object detection, 2nd stage uses Monte-Carlo sampling to estimate object 6 DoF pose.

around objects, with confidence scores and object class labels produced from any state-of-art object detection convolutional neural network (CNN). This input represents the object probability distribution over the observed scene. The CNN itself can be implemented by various network architectures such as VGG [18], ResNet [19] and Squeezenet [20] as discussed in [21]. However, the specific CNN architecture is not the focus of this paper. Below we describe the generative sampling algorithm in detail, generally following the design presented in [7].

Given an RGB-D observation (Z_r, Z_d) from a robot sensor Z_r for RGB image and Z_d for depth image, our goal is to maximize the conditional joint distribution $P(q, b|o, Z_r, Z_d)$ for each object where q is the 6 DoF pose for the object and o , b are the class label and bounding box respectively from the CNN output. The problem can be formulated as:

$$P(q, b|o, Z_r, Z_d) \quad (1)$$

$$= P(q|b, o, Z_r, Z_d)P(b|o, Z_r, Z_d) \quad (2)$$

$$= \underbrace{P(q|b, o, Z_d)}_{\text{pose estimation}} \underbrace{P(b|o, Z_r)}_{\text{detection}} \quad (3)$$

The second-stage takes the object detection results from the CNN and performs Monte-Carlo sampling via iterative likelihood weighting. In the initial stage, the algorithm generates a set of weighted **samples** $\{q^{(i)}, w^{(i)}, b^{(i)}, z^{(i)}\}_{i=1}^M$ to represent the belief of the object pose over the entire image. The value $q^{(i)}$ represents the 6 DoF pose of the sample object and $w^{(i)}$, $b^{(i)}$ and $z^{(i)}$ are associated with the probability, the bounding box of the object from the first stage, and the observed point cloud within the bounding box region, respectively. For each sample, given its object class o , pose $q^{(i)}$ and corresponding geometric model, the algorithm renders a 3D point cloud $r^{(i)}$

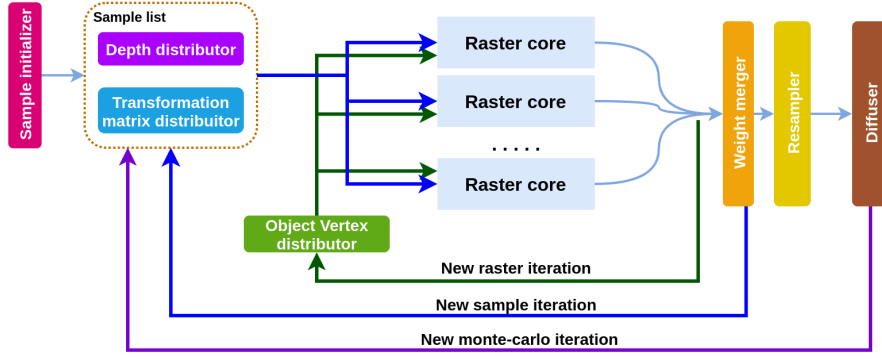


Fig. 2: FPGA diagram: Depth and transformation distributor sends a new pose and depth region associated with each sample to each raster core. Object vertex distributor transfers each triangle of the geometric model to raster. Each raster core works in parallel to calculate inlier score of samples at different poses.

of the sample using z-buffering of a 3D graphics engine. The weight $w^{(i)}$ of each sample is updated to estimate how close the sample matches the observation. We use a pixel-wise *inlier* function defined in Eqn. 4 to measure the matching between sample and observation:

$$\text{Inlier}(p, p') = \mathbf{I}(\|p - p'\|_2 < \epsilon), \quad (4)$$

where p, p' refers to a point in an observation point cloud $z^{(i)}$ and a point in a rendered point cloud $r^{(i)}$ from the sample pose, respectively. \mathbf{I} is the indicator function. An inlier is defined if a rendered point is within a certain distance threshold range ϵ from an observed point. The number of inliers is defined as:

$$N^{(i)} = \sum_{a \in z^{(i)}} \text{Inlier}(r^{(i)}(a), z^{(i)}(a)), \quad (5)$$

where a is an index of a point within z^i . We can use this value to obtain two raw-pixel inlier ratios: $N^{(i)}/N_b$, where N_b is the number of observation points within the bounding box $b^{(i)}$, and $N^{(i)}/N_r$, where N_r is the number of rendered points within the bounding box.

Next, using these ratios and probability c from the CNN, the weight w^i for each sample is computed as:

$$w^{(i)} = \alpha * \frac{N^{(i)}}{N_b} + \beta * \frac{N^{(i)}}{N_r} + \gamma * c, \quad (6)$$

where α, β, γ are the coefficients that are empirically determined and sum up to 1.

To get the optimal pose q^* , we follow the procedure of importance sampling [22] to assign a new weight to each sample. During this process, each sample pose, $q^{(i)}$, is diffused with a Gaussian distribution in the space of 6 DoF poses with a small δ to increase sample variance:

$$q^{(i)} = (x, y, z, \text{roll}, \text{pitch}, \text{yaw}) + \mathcal{N}(0, \delta). \quad (7)$$

Once the average sample weight is above a threshold, τ , we consider the algorithm converged and q^* will be selected as the sample with the highest weight w^* .

The most computationally expensive step in this process is the rendering and sample weight computation, which includes a pixel-wise inlier calculation. The amount of computation

and memory grows linearly with the number of samples we choose for the design. Even though modern GPUs can achieve high parallelism, the high energy consumption makes them less suitable for mobile platforms such as autonomous robots. In addition, their runtimes may still not allow for real-time operation. Our goal is to design various optimizations that can be implemented directly in hardware in order to achieve both faster runtime and reduced energy consumption.

IV. METHODOLOGY

Our FPGA implementation of Monte-Carlo sampling is illustrated in Fig. 2. The CNN object detection output consisting of probability and bounding box information is stored on off-chip memory and transferred to the second stage Monte-Carlo sampling module. The *sample initializer* generates N samples (*sample list* in Fig. 2) and the information for each sample (i.e., the 6DoF pose, bounding box region, and geometric model) is distributed to a *raster core* through a *transformation matrix distributor*, *depth distributor* and *object vertex distributor*. Each *raster core* performs rasterization and inlier comparison on a single sample at a time. The *weight merger* step will fetch the inlier scores from each raster core, calculate the weights for each sample, and send them to the *resampler*. After all the samples are processed, the *resampler* generates a new *sample list* of 6 DoF poses based on the weight of each sample. Finally, the *diffuser* stage adds Gaussian noise to each sample's 6 DoF pose. We then start a new *Monte-Carlo iteration* for the new sample list. We will next describe each of these steps in more detail in the following subsections.

A. Rasterizing

Rasterization is a process in computer graphics that converts a geometric model defined by vertices and faces to a raster image, defined by a series of pixels each with a depth value. The result of rasterization is an image of what a 3D object would look like at a certain view point. To implement rasterization in hardware, we designed a specialized *raster core* processing unit that pipelines the rasterization and inlier comparison steps for a given sample. The processing unit is illustrated in Fig. 3. At every *raster iteration*, a triangle from the geometric model is transformed with a sample's 6 DoF pose and rasterized, after which a depth value at each pixel within the triangle is

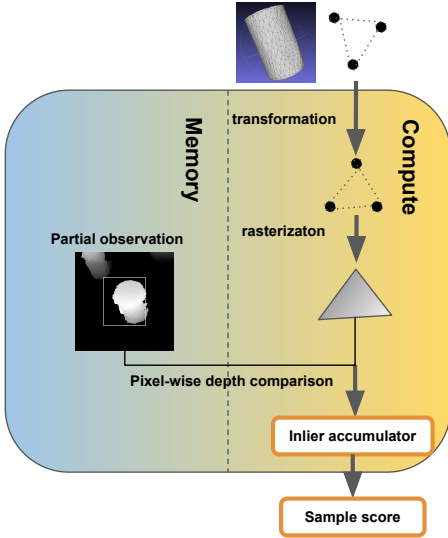


Fig. 3: Raster core flow: Memory unit stores a region in the observation depth defined by sample bounding box. Raster core operates on a single triangle at a time. The raster core pipelines the transformation, rasterization, and pixel-wise depth comparison between rasterized pixel and observation depth and outputs an inlier score.

calculated and compared to the observed depth region stored in the raster core. The comparison results are accumulated and output as an inlier score after all the triangles within the geometric model are processed. By pipelining these two steps, there is no need to store the rasterization result, and instead we only keep track of the inlier score from each sample.

We further reduce computational complexity and memory utilization by using partial rasterization. Note that since Eqn. (6) only pertains to the region within the bounding box, we only need to rasterize within this region. This partial rasterization is illustrated in Fig. 4. Backface culling is a standard algorithm inside a 3D graphic pipeline that removes the faces of the object model occluded by some other triangle [23]. For our purposes, we apply backface culling to reduce the total number of rasterized triangles by using the dot product between the surface normal and the camera point of view direction to judge if a face is occluded. On average, we found that we can reduce the number of rendered triangles by approximately 50% using this technique.

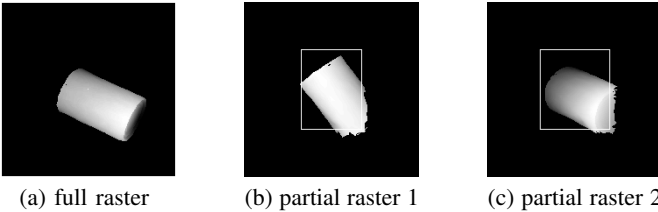


Fig. 4: Partial rendering: (a) full rasterization of the object, (b) and (c) partial rasterizations within the sample bounding boxes.

B. Inlier

In the original algorithm described in [7], a 3D point cloud is used to represent the rasterized sample and observation. In order to reduce the computation and memory overhead, we

wanted to modify the inlier comparison to a 1D depth representation. Given a point (x, y, z) in an observation pointcloud and a point (x', y', z') in a rendered point cloud the 3D Euclidean distance between two points can be computed noting that:

$$d = \sqrt{(x - x')^2 + (y - y')^2 + (z - z')^2}. \quad (8)$$

Given a depth z at pixel (p_x, p_y) , the x and y values can be calculated as:

$$\begin{aligned} x &= (p_x - C_x) \cdot z / f_x \\ y &= (p_y - C_y) \cdot z / f_y \end{aligned} \quad (9)$$

where C_x, C_y, f_x, f_y are camera intrinsic parameters (i.e., center offset and focal length). By substituting the x, y values in Eqn. (8) with the formulation in Eqn. (9), we see that for the same pixel (p_x, p_y) the distance differences in the x and y directions are proportional to the distance differences in the z direction. That is:

$$\begin{aligned} x - x' &= (p_x - C_x) \cdot z / f_x - (p_x - C_x) \cdot z' / f_x \\ &= (p_x - C_x) \cdot (z - z') / f_x \\ &\propto (z - z') \\ y - y' &= (p_y - C_y) \cdot z / f_y - (p_y - C_y) \cdot z' / f_y \\ &= (p_y - C_y) \cdot (z - z') / f_y \\ &\propto (z - z') \end{aligned} \quad (10)$$

Therefore, we can approximate the 3D Euclidean distance computation with a much simpler 1D depth comparison without affecting the pose estimation accuracy.

C. Depth Distributor

In the inlier calculation step, each rendered sample is compared with its corresponding observation depth region defined by a bounding box generated from the CNN output in the first stage. Therefore, each raster core must read a depth region from the entire depth image stored in on-board memory. Naively, if we distribute each region to each raster core in series, we need to repeatedly access the same memory location multiple times for the overlapping areas. Instead, we designed a depth distributor, as illustrated in Fig. 5, to reduce the amount of redundant memory accesses for overlapping depth regions. Our algorithm first divides the depth image into multiple sub-regions and identifies each region with a corresponding raster core number. The data from the overlapping regions are then read from memory once and distributed to multiple raster cores in parallel. In this way, the more overlapping areas we have among different regions, the faster the depth distribution can be completed. In particular, assuming that as more *Monte-Carlo iterations* are completed more samples will converge to the same bounding box, the runtime and power consumption of the depth distributor for later iterations will decrease.

D. Sorting and Resampling

To execute the resampling stage, we sort the samples by their corresponding weight. While sorting is not strictly required for importance sampling, we can sort the samples by their weights and only resample from the top $x\%$ to

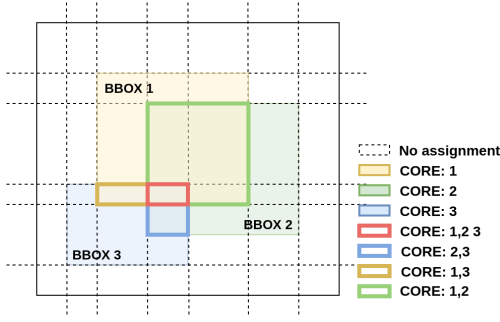


Fig. 5: Illustration of depth distribution. Each region shown within dotted lines is distributed to a raster core and overlapping regions (highlighted by the different colored boxes) will be distributed in parallel to multiple raster cores.

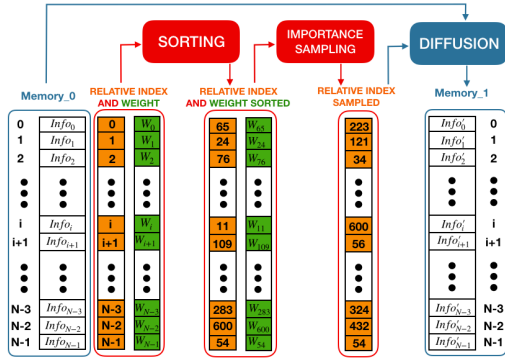


Fig. 6: Sorting and resampling. Starting with our N -entry sample list in *Memory_0*, we sort the entries by weight in ascending order. In this example, $W_{65} > W_{24} > W_{76}$, etc. *Memory_0* and *Memory_1* will be read and written alternately across each iteration.

further reduce resampling memory accesses. Sorting a sample generally requires moving all its object pose information (i.e., $x, y, z, roll, pitch, yaw$). However, to reduce data movement, we sort the index of the sample based only on the sample weight. Once the sorting step is complete, we conduct sampling to generate new sample indices.

Recall from Section III that resampling uses importance sampling to generate new samples from the sample weight distribution. We store the cumulative density function (CDF) of the normalized sample weight x in an array, $\Phi(x)$. A random number $r \in U(0, 1)$ is generated and compared with values in the CDF array until we find the first sample i where $r < \Phi(x^{(i)})$. The number of memory reads to the CDF array increases as $r \rightarrow 1$, since the memory locations must be analyzed starting from the zero cell up to the desired one. To reduce memory accesses, we use a separate memory to store a set of threshold values $\{t_0, t_1, \dots, t_n\}$ with a constant step, where $t_i \in [0, 1]$ such that we first execute a coarse-grained search to find region $[t_k, t_{k+1}]$ where r falls, and then do a fine-grained search for i where $\Phi(x^{(i)}) \in [t_k, t_{k+1}]$. From our experiments, we found that this technique greatly reduces the average number of memory accesses from 410 to 10.

To further speed up execution time, we implemented a ping-pong buffer for the diffusion stage, as illustrated in Fig. 6. We alternate fetching sample information from either *Memory_0*

or *Memory_1* using the new sample indices generated from the resampler, add Gaussian noise to the 6 DoF pose, and save the new samples in the opposite memory buffer.

V. EXPERIMENTAL RESULTS

We implemented our Monte-Carlo sampling algorithm on a Xilinx[®] Virtex UltraScale FPGA ZCU102 board using the Vivado HLS high-level synthesis tool. Given the memory and computational resources of this board, we can fit a total of 20 raster cores in our design. In general, the more samples we use, the better we can approximate sample distribution. In our case, we chose to process a total of 620 samples because it is sufficient to describe our algorithm search space, so 31 sample iterations are needed to render all the samples.

We compared runtime, power, energy consumption, and accuracy of our FPGA implementation to a CPU-GPU hybrid reference design implemented on two platforms: 1) an Nvidia[®] Titan Xp with and Intel Xeon E5, and 2) an Nvidia[®] Jetson TX2 with a quad-core ARM A57. We note that both GPU platforms are more powerful than our FPGA in terms of memory capacity, compute resources, and clock frequency. Sample initialization, resampling, and diffusion are done on the CPU since their operations are sequential in nature, while sample rendering and inlier computation is done on the GPU. We use OpenGL to render all the samples and program the CUDA cores to perform the inlier computation. We create a kernel where every pixel distance comparison is assigned to a CUDA thread and processed concurrently. Since the GPU has high memory bandwidth, we keep one copy of the observation depth in the GPU memory such that each sample accesses the observation depth to compute the inlier.

The dataset used in the experiments contains scenes collected by a Kinect RGBD camera with objects from the YCB dataset [8]. Each scene captures a depth image of size 640 X 480. In each scene, 5–7 different objects are placed on a table. An example scene is shown in Fig. 7. We choose to test these 5 different objects for their different sizes and symmetries.

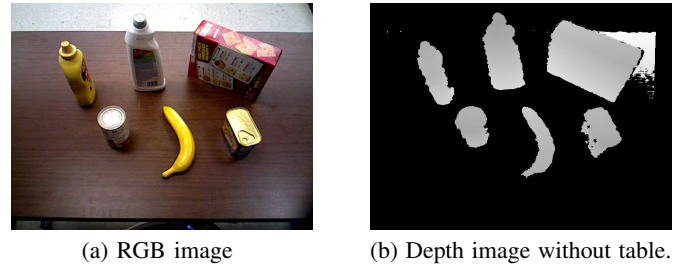


Fig. 7: A test scene containing objects from the YCB dataset.

A. Runtime

Table I reports the average runtime for the rendering and inlier stages on both FPGA and GPU platforms. Note that the FPGA implementation has a faster runtime than the Jetson version and slightly slower runtime compared with Titan.

	banana	cracker box	potted meat can	mustard bottle	tomato soup can
FPGA	17.08ms	24.70ms	18.67ms	18.01ms	19.32ms
Titan	10.24ms	15.78ms	11.29ms	12.52ms	11.66 ms
Jetson	188.81ms	244.61ms	192.45ms	205.53ms	193.66ms

TABLE I: Average runtimes for render+inlier process: FPGA at **200MHz**, Titan Xp at **1.4GHz** and Jetson TX2 at **1.3GHz**

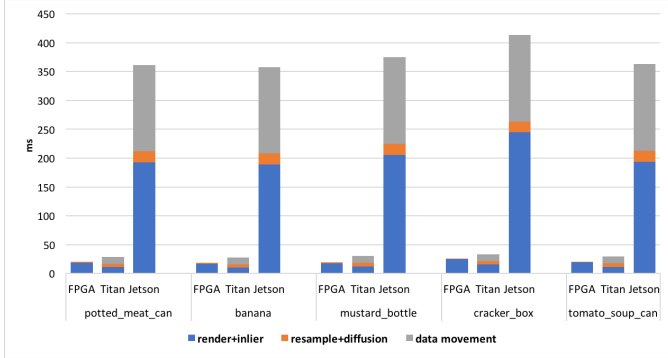


Fig. 8: Average runtimes for one Monte-Carlo iteration.

In Fig. 8 we show the average runtime per entire *Monte-Carlo iteration*, broken down into three stages: 1) rendering+inlier computation, 2) data transfer between GPU and CPU, and 3) resampling and diffusion computation. Since our FPGA implementation keeps the entire Monte-Carlo sampling algorithm on board, the total data transfer time is greatly reduced and thus has advantages for per-iteration runtime.

Note that while Monte-Carlo inference processes each object in series, the robot can start object manipulation as part of a pick-and-place action as soon as the first object completes. On average, it takes 50 *Monte-Carlo iterations* for the algorithm to converge for each object. Given the average runtimes from Table I, our FPGA implementation can process a single object in approximately 1 second. Since a robot movement can take a few seconds to complete, we can start to pick the next object without stalling the robot action; thus, we consider 1 second as real time processing for this application.

Finally, the benefit of our depth distribution implementation is shown in Fig. 9. Here we chose to test average runtime per iteration for 50 iterations. Note that the average runtime decreases over the iterations as the objects converge.

B. Resource and Power

The resource utilization for each stage of our algorithm is shown in Table II and the power and energy consumption of each implementation is shown in Table III. The FPGA power is collected from the Vivado[®] power analyzer, while the Titan power is estimated through the Nvidia[®] Management Library, and Jetson power is measured through an on-board power monitor. Note that for our FPGA implementation, resampling and diffusion steps are responsible for less than 13% of total power/energy. In addition, the Titan and Jetson GPU only performs rendering and inlier computations and not resampling and diffusion stages since these are on the CPU. Just focusing on rendering+inlier power and energy, we see that the FPGA implementation is 33X more power efficient and 21X more

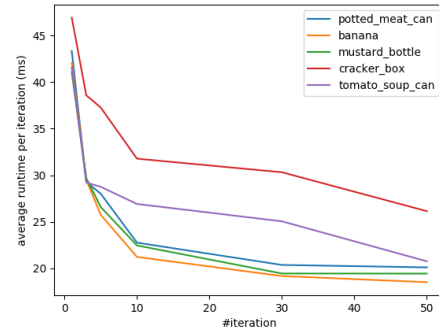


Fig. 9: FPGA per Monte-Carlo iteration runtime vs. iteration count.

energy efficient than the Titan Xp. Compared against the low power Jetson, our FPGA implementation is slightly more power efficient and 12X more energy efficient. Moreover, our solution is scalable to any FPGA platform by choosing the number of raster cores to balance the runtime versus resource and power tradeoff.

	BRAM36s	DSP48E2	LUT	FFs
sample initializer	0	6	2123	2409
20 raster cores	480	920	146000	172780
resample	1	10	926	855
diffuse	0.5	13	3588	1990

TABLE II: Resource utilization for each stage

	FPGA whole flow	FPGA render+inlier	Titan render+inlier	Jetson render+inlier
P_{ave}	3.85W	3.36W	110.34W	3.78W
E_{ave}	75.32mJ	65.70mJ	1357.12mJ	775.62mJ

TABLE III: Average power and energy consumption

C. Accuracy

To evaluate the accuracy of our implementation, we chose 5 objects and 9 different scenes, where each object occurs 5 times in the scene. We ran our algorithm 5 times for each scene to avoid randomness in the result. We use the average distance metrics ADD and ADD-S as defined in [8] to calculate the point distance error between predicted pose and ground truth pose for symmetric and non-symmetric objects respectively. Both GPU and FPGA implementations achieve around 52% pose estimation accuracy under an ADD threshold of 4 cm. We see that even though we simplified the inlier calculation and rasterization steps, the FPGA implementation achieves similar accuracy as the GPU implementation.

VI. CONCLUSIONS

In this paper, we have shown an effective hardware implementation of a Monte-Carlo sampling algorithm, as part of the two-stage discriminative-generative method used for pose estimation for robot manipulation. With our FPGA implementation, we are able to achieve real time performance with significantly reduced energy consumption, compared to either a high-performance or low power GPU implementation. Future work will consider adding a deep pipelined feature extraction step, along with rasterization, to provide higher accuracy for objects pose estimation in more clustered environments.

REFERENCES

- [1] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929-1958, January 2014. doi:10.5555/2627435.2670313.
- [2] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *CoRR*, abs/1412.6572, 2015.
- [3] Kevin Eykholt, Ivan Evtimov, Earlene Fernandes, Bo Li, Amir Rahmati, Chaowei Xiao, Atul Prakash, Tadayoshi Kohno, and Dawn Song. Robust physical-world attacks on deep learning visual classification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1625–1634, 2018. doi:10.1109/CVPR.2018.00175.
- [4] Zhiqiang Sui, Zheming Zhou, Zhen Zeng, and Odest Chadwicke Jenkins. SUM: Sequential scene understanding and manipulation. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3281–3288. IEEE, 2017. doi: 10.1109/IROS.2017.8206164.
- [5] Venkatraman Narayanan and Maxim Likhachev. Discriminatively-guided deliberative perception for pose estimation of multiple 3d object instances. In *Robotics: Science and Systems*, 2016. doi:10.15607/RSS.2016.XII.023.
- [6] Xiaotong Chen, Rui Chen, Zhiqiang Sui, Zhefan Ye, Yanqi Liu, R. Iris Bahar, and Odest Chadwicke Jenkins. GRIP: Generative robust inference and perception for semantic robot manipulation in adversarial environments. *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2019. doi:10.1145/3240765.3243493.
- [7] Zhiqiang Sui, Zhefan Ye, and Odest Chadwicke Jenkins. Never mind the bounding boxes, here’s the SAND filters. *arXiv:1808.04969*, 2018.
- [8] Yu Xiang, Tanner Schmidt, Venkatraman Narayanan, and Dieter Fox. PoseCNN: A convolutional neural network for 6d object pose estimation in cluttered scenes. *Robotics: Science and Systems XIV*, 2018. doi:10.15607/RSS.2018.XIV.019.
- [9] Jonathan Tremblay, Thang To, Balakumar Sundaralingam, Yu Xiang, Dieter Fox, and Stan Birchfield. Deep object pose estimation for semantic robotic grasping of household objects. *arXiv:1809.10790*, 2018.
- [10] Chen Wang, Danfei Xu, Yuke Zhu, Roberto Martn-Martn, Cewu Lu, Li Fei-Fei, and Silvio Savarese. Densefusion: 6D object pose estimation by iterative dense fusion. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3338–3347, 2019. doi:10.1109/CVPR.2019.00346.
- [11] Lszl Schffer, Zoltn Kincses, and Szilveszter Pletl. A real-time pose estimation algorithm based on FPGA and sensor fusion. In *International Symposium on Intelligent Systems and Informatics (SISY)*, Sep. 2018. doi:10.1109/SISY.2018.8524610.
- [12] Michael Schaeferling, Ulrich Hornung, and Gundolf Kiefer. Object recognition and pose estimation on embedded hardware: Surf-based system designs accelerated by FPGA logic. *International Journal of Reconfigurable Computing*, 2012:6, 2012. doi:10.1155/2012/368351.
- [13] Ryo Konomura and Koichi Hori. FPGA-based 6-DoF pose estimation with a monocular camera using non co-planer marker and application on micro quadcopter. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4250–4257, 2016. doi: 10.1109/IROS.2016.7759626.
- [14] Atsutake Kosuge, Keisuke Yamamoto, Yukinori Akamine, Taizo Yamawaki, and Takashi Oshima. A 4.8x faster FPGA-based iterative closest point accelerator for object pose estimation of picking robot applications. In *IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 331–331, April 2019. doi:10.1109/FCCM.2019.00072.
- [15] Riku Murai, Paul Kelly, Sajad Saeedi, and Andrew Davison. Visual odometry using a focal-plane sensor-processor. 2019.
- [16] Fynn Schwiegelshohn, Eugen Ossovski, and Michael Hübner. A fully parallel particle filter architecture for FPGAs. In Kentaro Sano, Dimitrios Soudris, Michael Hübner, and Pedro C. Diniz, editors, *Applied Reconfigurable Computing*, pages 91–102, Cham, 2015. Springer International Publishing.
- [17] B. G. Sileshi, J. Oliver, and C. Ferrer. Accelerating particle filter on FPGA. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 591–594, 2016. doi: 10.1109/ISVLSI.2016.66.
- [18] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv:1409.1556*, 2014.
- [19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.
- [20] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and 0.5 mb model size. *arXiv:1602.07360*, 2016.
- [21] Yanqi Liu, Alessandro Costantini, Zhiqiang Sui, Zhefan Ye, Shiyang Lu, Odest Chadwicke Jenkins, and R. Iris Bahar. Robust object estimation using generative-discriminative inference for secure robotics applications. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2018. doi:10.1145/3240765.3243493.
- [22] Malvin H. Kalos and Paula A. Whitlock. *Monte Carlo Methods. Vol. 1: Basics*. Wiley-Interscience, USA, 1986.
- [23] Subodh Kumar, Dinesh Manocha, Bill Garrett, and Ming Lin. Hierarchical back-face culling. In *7th Eurographics Workshop on Rendering*, pages 231–240, 1996.