GPU based approach for fast generation of robot capability representations

Enfoque basado en GPU para la generación rápida de representaciones de capacidad de robot

Daniel García-Vaglio[1], Federico Ruiz-Ugalde[2]

Hernández-Zamora, M.F. Gpu based approach for fast generation of robot capability representations. *Tecnología en Marcha*. Tecnología en marcha. Vol. 35, special issue. November, 2022. International Work Conference on Bioinspired Intelligence. Pág. 50-57.

https://doi.org/10.18845/tm.v35i8.6449

1 Electrical Engineering Department. University of Costa Rica. Costa Rica
2 Electrical Engineering Department. University of Costa Rica. Costa Rica.
E-mail: federico.ruizugalde@ucr.ac.cr

Keywords

GPU computation; capability maps; robotic dexterity.

Abstract

Capability maps are an important tool for enabling robots to understand their bodies by providing a way of representing the dexterity of their arms. They are usually treated as static data structures be- cause of how computationally intensive they are to generate. We present a method for generating capability maps taking advantage of the parallelization that modern GPUs offer such that these maps are generated approximately 50 times faster than previous implementations. This system could be used in situations were the robot has to generate this maps fast, for example when using unknown tools.

Palabras clave

Cálculo de GPU; mapas de capacidad; destreza robótica.

Resumen

Los mapas de capacidad son una herramienta importante para permitir que los robots comprendan sus cuerpos al proporcionar una forma de representar la destreza de sus brazos. Por lo general, se tratan como estructuras de datos estáticas debido a lo intensidad en computación que pueden generar. Presentamos un método para generar mapas de capacidad aprovechando la paralelización que ofrecen las GPU modernas, de modo que estos mapas se generan aproximadamente 50 veces más rápido que las implementaciones anteriores. Este sistema podría utilizarse en situaciones en las que el robot tiene que generar estos mapas rápidamente, por ejemplo, cuando se utilizan herramientas desconocidas.

Introduction

One of the challenges of modern robotics is to build robots that can work as competent autonomous assistants for humans, such that we can share the same workspaces. It is necessary to enable robots to understand their bodies and their capabilities. This would enable robots to reason about how to do certain tasks and whether or not they need tools or help from other agents to complete them. In particular we are interested on representing the dexterity of a robot, to enable it to eventually reason about the best ways to manipulate objects (find strategies for optimizing dexterity) [10,7].

One approach for representing robotic dexterity are capability maps. These are maps that assign a reachability measure to every point in space for a robotic arm; that is, how many ways the robot is able to reach a certain point in space [8]. Figure 1b shows an example of the capability map of the KUKA LWR 4+ arm (figure 1a). These maps have multiple application that will be discussed in section 2.

One of the drawbacks of capability maps is that they are very computation- ally intensive to generate, so they are normally calculated once and treated as static data structures [5,2]. But this is not always the most convenient approach. When designing a robotic arm, it is useful to see how changing certain parameters would affect the capabilities of the arm. Having rapid feedback can speed-up the design process. Another example were capability maps could change are if the arm of a robot changes while executing tasks. If a joint gets damaged and the robot decides that the best strategy is to lock it in-place to avoid further damages, the capability

map changes. Also, when the robot is using a tool we are interested on the tip of the tool rather than the tip of the arm. So, whenever the robot picks a new previously unknown tool, it is possible that it will require to generate a new capability map that takes into consideration the effects of the tool.

As shown previously, there are certain situations where we would need to generate capability maps fast. In this paper we present a method for generating them using the parallel computation capabilities of modern GPUs and show how this approach differs from previous implementations.

Related work

Since very early in the development of redundant robotic manipulators, there has been a discussion about how to measure dexterity. One of the proposed dexterity measures is to analyze the amount of orientations that a robot can reach at the point of interest [4]. This idea was then expanded into capability maps, which are maps that represent how many orientations a robotic arm is able reach in every point of its workspace [11]. The points where the robot is able to reach many orientations are considered points of high dexterity, while points that are only reached with one orientation are considered points of low dexterity.

The first generation approaches only took into consideration using inverse kinematics for deciding if a pose was reachable or not [11]. This generation method was very slow. To increase performance a method that used forward kinematics was introduced, but this meant an accuracy penalty [9].

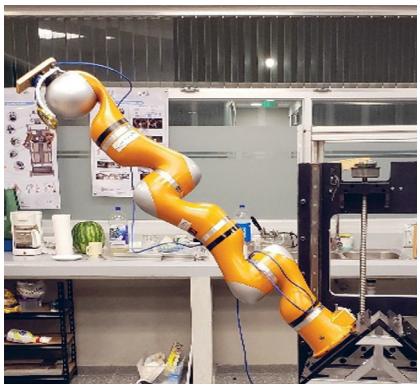
Capability maps have been used in a variety of applications. They have been used in planning, for deciding where to position the body of a robot in front of a table to optimize its manipulation capabilities [6], they have also been used to understand dual-arm manipulation, both for control [5] and for designing hardware [1].

There is a previous example of generating the workspace of a robotic arm with a GPU based approach. They were able to generate the workspace in less than a second [3]. The workspace is the set of positions that can be reached by the robot, but does not encode how well they are reached. They generated random robot poses and incorporated them into a point-cloud to build the workspace. This problem is easier because they don't have to compute how well a point is reached, so they only need to reach each point once, but when generating a capability map each point has to be visited multiple times (from 500 to 6000 in the case of our experiments).

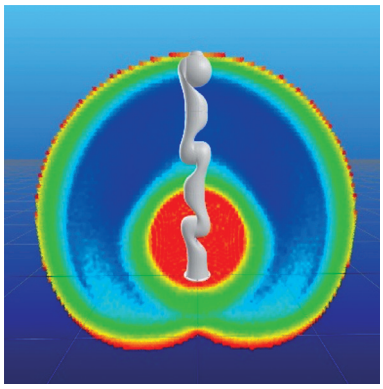
System Overview

The first step is to compute the hierarchical subdivision of space to be able to do the analysis [8]. Let C be the maximum reach length of the arm, we only take into consideration the cube centered on the arm origin of size $2C \times 2C \times 2C$. Each dimension is divided into N_c equal segments, giving a total of N_c^3 voxel. For generating the orientations we used the Euler angles intrinsic convention, also known as yaw, pitch and roll. For generating the pitch and yaw we considered the problem of dividing a sphere's surface into N_s equal areas. This was achieved by creating a Fibonacci spiral where the n_s th section center is given by (1), with φ being the golden ratio, as shown in figure 1c. And finally the roll is generated by dividing 2π into N_r equal parts.

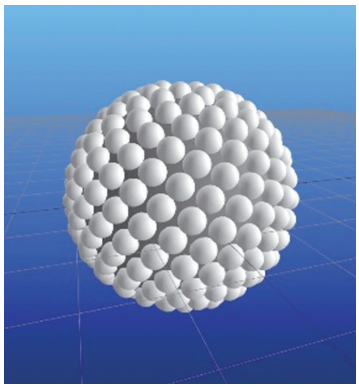
$$\theta_{yaw} = 2\pi\,(2 - \varphi)\,n_s, \theta_{pitch} = \arcsin\left(-1+2\,\frac{n_s}{N_s}\right) \qquad (1)$$



(a) Real kuka arm in our robot for reference



(b) Capability map generated from over 109 random poses. Red is low reachability and blue high reachability



(c) Distribution of 100 points using the fibonacci spiral

Figure 1. Data generation examples.

There are three techniques to compute the scores for the voxels to create the capability map, use only forward kinematics, only inverse kinematics or the hybrid approach. Because we are seeking a very fast generation of the map, the pure forward kinematics is used. There is an accuracy penalty (11.45% loss in the worst case) that is paid for using only forward kinematics [9].

It is necessary to generate random poses of the end-effector, for this purpose we generate a random number for each joint from a uniform distribution inside the joints limits and it is used as the joint state. Then the forward kinematics algorithm is used to compute the pose of the end effector. These poses are in- dependent from one another, therefore this is done in parallel in the GPU. The parallel forward kinematics solver was implemented from scratch. These is done by generating the homogenous matrices that represent the transformations of each joint and link of the kinematic chain that model the arm and multiplying them in order [9].

The next step is to build the reachability map, this is to find the voxel in which the end-effector is, and which orientation from the generated ones is the closest. Although the reachability map is a 6-dimensional structure (one dimension for each degree of freedom), we find it more efficient to represent it as a 2D matrix called reachability matrix. There is a row for each possible position, and a column for each possible orientation. Then the reachability map is represented by writing a 1 (or a True) in the (c_p,c_θ) coordinates of the matrix (where cp is the position of the voxel containing the end effector, and c_θ the closest orientation). So, a 1 means that the position and orientation was reached, and a 0 means that it wasn't.

For finding the position and orientation we do not do a normal search, because it is very slow, instead we take advantage of the fact that we know how the voxels and the orientations were generated to find the solution faster. The coordinate inside the reachability matrix for the row (position) is given by (2) where r()˙ is the round function, s is the step between voxel centers, (x,y,z) is the exact position of the end-efector, and (x_0,y_0,z_0) is the position of the first voxel.

$$c_p = r\left(\frac{z - z_0}{s}\right) + N_c r\left(\frac{y - y_0}{s}\right) + N_c^2 r\left(\frac{z - z_0}{s}\right) \quad (2)$$

We are using 3 x 3 rotation matrices for representing rotations, because of how well they work with forward kinematics. Let *M* be the matrix that represents the orientation of the end-effector, we know because of the intrinsic yaw-pitchroll that the entry *M*(2,0) represents the sine of the

pitch -sin(θ_{pitch}). Then solving for ns in (1) one knows that C_θ is between C_s and C_s + N_r where C_s= $[[\frac{N_s}{2}(1\text{-}M(2,0))]]$. This means that for finding the nearest orientation we have reduced the search to a list of orientation of size N_r, instead of N_s x N_r.

To find the closest orientation we need to calculate the angle between each orientation and the resulting orientation of the end-effector. Let *A* and *B* be two rotation matrices, the angle between them is given by arccos ((T_r (AB^T)-1)/2). The arccos function is very expensive to compute, so to avoid that we use the fact that it is monotonous in [0,π], therefore order relations prevail when using other functions (that are also monotonous). So, instead we calculate α=1- (T_r (AB^T)-1)/2, and then take the orientation that gives the minimal α. Doing this optimization almost duplicated the speed of the computation.

Filling the reachability matrix is done in parallel for each generated random end-effector pose. The process is completely independent until the reachability matrix has to be written. The advantage is that this is a write-only data structure for this part of the algorithm. The pose generation process and filling the reachability matrix was implemented in a single CUDA kernel of size 1024 x 1024 (blocks per grid, and threads per block respectively). It gives a total of 1048576 threads, each one with a different random end-efector pose. This are not enough poses to make sure each voxel was visited the right amount of times, so we have to execute the kernel NB times, where N_B $[\frac{6}{6250000} N_c^3 N_s N_r]$. There is a linear complexity in regards to the amount of voxels and N_s, but a quadratic complexity in regards to N_r.

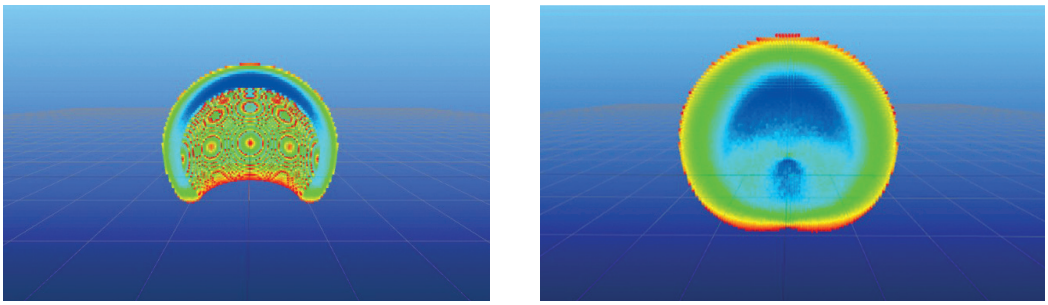
The last step is to calculate the capability map from the reachability matrix. This is done by calculating the sum of each row of the reachability matrix. The parallelization of this step is trivial, and it was implemented in a separate kernel.

Once the summation is done there is a score for each voxel. The scores are then and converted into the colour scale (red for 0 and blue for 1).

Results

We ran our system in a GeForce GTX 1060 with 1280 CUDA cores and 6GB of dedicated video memory. It was implemented with the CUDA framework using the numba just in time compiler for Python3.8. The experiments consisted of generating capability maps with different N_c, N_s and N_r values, for the KUKA LWR 4+ robotic arm.

The first results that should be discussed is how the capability map can change while the robot is executing manipulation tasks. Figure 2a shows the capability map of the KUKA LWR 4+ robotic arm if the fourth joint (commonly known as elbow) gets locked at 0o, it is different to the normal capability map presented in figure 1b. Because there is no elbow, the workspace is limited to a hollow shape, almost like the outer layer of the normal capability map. It is clear that if a joint has to be locked the normal capability map must not be used anymore and the robot is required to calculate the new capability map. Figure 2b shows the capability map of the arm if a 30cm tool is attached to the end of the arm, not only the capability map grows in size, but the distribution of high capability regions changes completely. This shows why it is important to generate a new capability map when a new tool is being used. In both cases the capability map must be generated fast so that the robot can continue executing tasks.

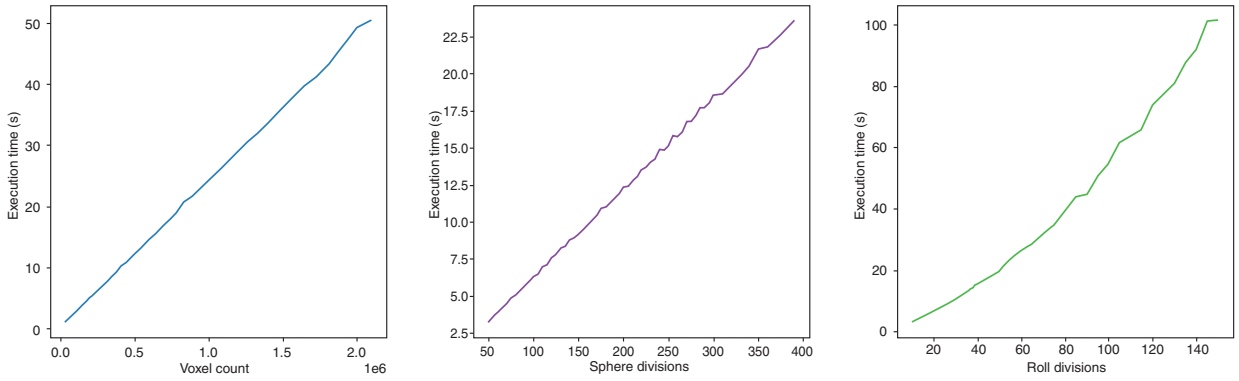


(a) 4th joint (the elbow) is locked at 0

(b) 30cm tool attached at the end of the arm

Figure 2. Capability maps of the KUKA LWR 4+ in different kinematic configurations

Although the main objective is to give robots the capacity to quickly generate capability maps during task execution the GPU approach could also be used to generate detailed maps with very high resolution. For this end, it is necessary to analyze how the system behaves as the resolution grows. In figures 3a, 3b and 3c we show how the execution time behaves as the voxel count (N_c^3), N_s and N_r grow. This graphs were generated by leaving the other two parameters constant at N_c= 50, N_s = 50, and N_r = 10. The first two show a linear behaviour while the last one shows a quadratic behaviour as predicted in section 3. The main limitation that is not letting the capability map grow is memory. The reachability matrix has to be stored in GPU memory so that the algorithm can run fast, then the upper bound is the GPU memory, which in our case was 6GB.



(a) Execution time against voxel count

(b) Execution time against N_s

(c) Execution time against N_r

Figure 3. Behaviour of execution time against different workspace subdivision parameters

Finally, our system should be compared to other implementations of capability maps generators. Most literature is focused on using capability maps, but not in generating them. From the papers that discuss the problem of generating them, the authors in [9] explained optimization strategies for generating this structures. They offered the fastest generation times that we could find in the literature, but they were running on CPUs. As we show in table 1 we were able to generate capability maps around 50 times faster once the algorithm was implemented for GPU. We offer a comparison to all the resolution configurations that produce a reachability matrix that fits in our GPU's memory, but in all cases our implementation was considerably faster.

Conclusions and future work

We presented a method for fast generation of capability maps that could be used in situations were the robot needs to calculate them because of changes in the kinematic configuration of its arms. This could be due to damages in a joint that forces it to get locked in-place, or if the robot takes an unknown tool that changes its distribution of dexterity in space. We were able to generate capability maps around 50 times faster than previous implementations which reduced the generation time from the order of minutes to the order of seconds.

Table 1. Comparison between a previous implementation and ours.

N_c	N_s	N_r	Previous	Ours	Speed-up
32	50	10	49.8	1.07	46.54
	100	20	199.8	3.86	51.76
	200	300	552.6	12.02	45.97
47	50	10	139.8	2.71	51.59
	100	20	615.6	10.88	56.58
	200	300	1734.6	36.19	47.65
94	50	100	1264.8	20.47	61.78

This speed increment was achieved because the algorithm was implemented for the GPU instead of the CPU, which is better suited for data parallel problems. Now that vision algorithms are increasingly complex, GPUs are becoming more common in robotic platforms. This enables roboticists to add more algorithms that depend on this hardware like the one presented in this paper. One drawback is that while the capability map is being generated the GPU is utilized at 100% for most of the generation process, so complex vision processes would not be able to run for that period of time.

The biggest limitation is memory. The reachability matrix consumes a lot of space as the resolution (Ns, Nc and Nr) increases. Because the end-effector poses are generated at random from a uniform distribution at the joint-space, it is necessary to have the entire matrix loaded in GPU memory. One part of the future work is to find ways for allowing the reachability matrix grow in a way that performance doesn't get too negatively impacted but better resolutions are also possible.

This project is part of our efforts of building a humanoid robot to assist people (see figure 1a) in real-life scenarios. So, another next step is to incorporate this project into a our Humanoid robot cognitive system. This would allow us to test how fast generation of capability maps augment the manipulation capabilities of the robot by enabling it to generate new capability maps "on the fly" for the tools it uses.

Acknowledgment

The present paper is a partial result of a research project entitled "Manipulation of everyday objects using an object model system. Creation of an object model library and a new object model, 322-B6-279" funded by the Research Agency of the University of Costa Rica (Vicerrectoría de Investigación). Additional funding and support is provided by the University of Costa Rica

Dean's office, Electrical Engineering department, Graduate Program in Electrical Engineering, Engineering Research Institute and by the Institute for Artificial Intelligence at the University of Bremen.

References

[1] Chaves-Arbaiza, I., García-Vaglio, D., Ruiz-Ugalde, F.: Smart Placement of a Two- Arm Assembly for An Everyday Object Manipulation Humanoid Robot Based on Capability Maps. In: 2018 IEEE International Work Conference on Bioinspired Intelligence (IWOBI), pp. 1–9 (2018). DOI 10.1109/IWOBI.2018.8464192

[2] Forstenhäusler, M., Wetner, T., Dietmayer, K.: Optimized Mobile Robot Posi- tioning for better Utilization of the Workspace of an attached Manipulator. In: 2020 IEEE/ASME International Conference on Advanced Intelligent Mechatronics (AIM), pp. 2074–2079 (2020). DOI 10.1109/AIM43001.2020.9158922. ISSN: 2159-6255

[3] Jauer, P., Kuhlemann, I., Ernst, F., Schweikard, A.: GPU-based real-time 3D workspace generation of arbitrary serial manipulators. In: 2016 2nd International Conference on Control, Automation and Robotics (ICCAR), pp. 56–61 (2016). DOI 10.1109/ICCAR.2016.7486698

[4] Klein, C.A., Blaho, B.E.: Dexterity Measures for the Design and Control of Kinematically Redundant Manipulators. The International Journal of Robotics Research 6(2), 72–83 (1987). DOI 10.1177/027836498700600206. URL http://journals.sagepub.com/doi/10.1177/027836498700600206

[5] Liu, Q., Chen, C.Y., Wang, C., Wang, W.: Common workspace analysis for a dual-arm robot based on reachability. In: 2017 IEEE International Conference on Cybernetics and Intelligent Systems (CIS) and IEEE Conference on Robotics, Automation and Mechatronics (RAM), pp. 797–802 (2017). DOI 10.1109/ICCIS. 2017.8274881. ISSN: 2326-8239

[6] Makhal, A., Goins, A.K.: Reuleaux: Robot Base Placement by Reachability Analy- sis. In: 2018 Second IEEE International Conference on Robotic Computing (IRC), pp. 137–142 (2018). DOI 10.1109/IRC.2018.00028

[7] Morgan, A.S., Hang, K., Bircher, W.G., Alladkani, F.M., Gandhi, A., Calli, B., Dollar, A.M.: Benchmarking Cluttered Robot Pick-and-Place Manipulation With the Box and Blocks Test. IEEE Robotics and Automation Letters 5(2), 454–461 (2020). DOI 10.1109/LRA.2019.2961053. Conference Name: IEEE Robotics and Automation Letters

[8] Porges, O., Lampariello, R., Artigas, J., Wedler, A., Borst, C., Roa, M.A.: Reach- ability and dexterity: Analysis and applications for space robotics. In: Proceedings of the Workshop on Advanced Space Technologies for Robotics and Automation (ASTRA) (2015)

[9] Porges, O., Stouraitis, T., Borst, C., Roa, M.A.: Reachability and Capability Analysis for Manipulation Tasks. In: M.A. Armada, A. Sanfeliu, M. Ferre (eds.) ROBOT2013: First Iberian Robotics Conference, Advances in Intelligent Systems and Computing, pp. 703–718. Springer International Publishing, Cham (2014). DOI 10.1007/978-3-319-03653-3_50

[10] Ruiz, E., Mayol-Cuevas, W.: Where can i do this? Geometric Affordances from a Single Example with the Interaction Tensor. In: 2018 IEEE International Confer- ence on Robotics and Automation (ICRA), pp. 2192–2199. IEEE, Brisbane, QLD (2018). DOI 10.1109/ICRA.2018.8462835. URL https://ieeexplore.ieee.org/ document/8462835/

[11] Zacharias, F., Borst, C., Hirzinger, G.: Capturing robot workspace structure: repre- senting robot capabilities. In: 2007 IEEE/RSJ International Conference on Intelli- gent Robots and Systems, pp. 3229–3236 (2007). DOI 10.1109/IROS.2007.4399105. ISSN: 2153-0866

[17] Chang, C.C., Lin, C.J.: LIBSVM: A library for support vector machines. ACM Transactions on Intelligent Systems and Technology 2, 27:1–27:27 (2011). Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm

[18] Anguita, D., Ghio, A., Ridella, S., Sterpi, D.: K-fold cross validation for error rate estimate in support vector machines. In: R. Stahlbock, S.F. Crone, S. Lessmann (eds.) Proceedings of the 2009 International Conference on Data Mining, DMIN 2009, July 13-16, 2009, Las Vegas, USA, pp. 291–297. CSREA Press (2009)