

3D Modeling, Distance and Gradient Computation for Motion Planning: A Direct GPGPU Approach

René Wagner

Udo Frese

Berthold Bäuml

Abstract—The Kinect sensor and KinectFusion algorithm have revolutionized environment modeling. We bring these advances to optimization-based motion planning by computing the obstacle and self-collision avoidance objective functions and their gradients directly from the KinectFusion model on the GPU without ever transferring any model to the CPU. Based on this, we implement a proof-of-concept motion planner which we validate in an experiment with a 19-DOF humanoid robot using real data from a tabletop work space. The summed-up time from taking the first look at the scene until the planned path avoiding an obstacle on the table is executed is only three seconds.

I. INTRODUCTION

For truly autonomous operation of complex humanoid robots in unknown, cluttered environments it is crucial to close the sensor-model-planning-action loop efficiently. This requires (a) the real-time self-acquisition of highly-detailed, highly-accurate 3D environment models and (b) efficient motion planning on these models. There have been recent breakthroughs in both areas: (a) The introduction of the Kinect sensor and the KinectFusion algorithm [13] have made 3D environment models of unprecedented quality available at an unprecedented frame rate (30Hz) and (b) recent advances in optimization-based motion planning (OMP) [17, 20] allow for the direct computation of smooth trajectories in reaching motions for dexterous manipulation and in dynamically optimal whole body motions [1]. In moderately cluttered scenes, stochastic OMP variants [10] are similarly robust [20] as global sample-based planners (e.g., RRT [11]), without having the drawback of the latter, which can primarily generate only jerky trajectories.

In this paper, we show that the model representation used by KinectFusion directly contains all information required for planning, how this can be used to compute obstacle avoidance and self-collision avoidance objective functions and their gradients directly on the GPU without ever transferring any model to the CPU. The evaluation of the objective and gradient functions dominate the computation time of any OMP. We validate our GPGPU implementation against an independently developed Mathematica (double precision, CPU) implementation and in an experiment with DLR’s 19-DOF humanoid robot Agile Justin [1] using real sensor data. Our focus is on the efficient objective and gradient

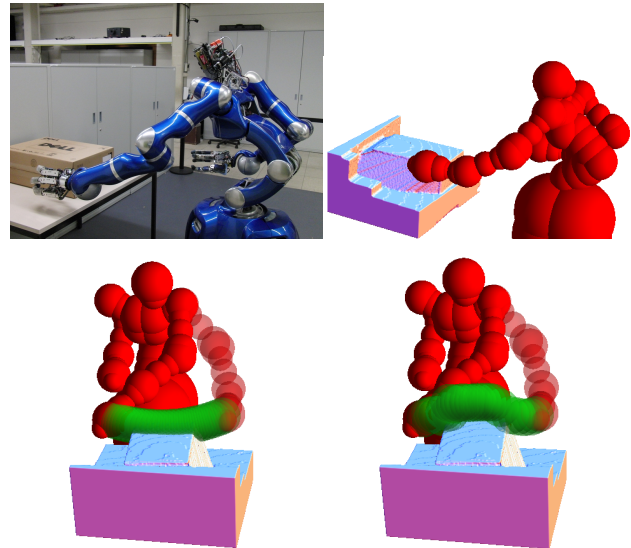


Fig. 1. DLR’s humanoid Agile Justin [1] in our evaluation scenario (top left): Its left hand is to be moved from the left of a cardboard box to the right of it. A 3D model of the scene is generated from depth data of a head-mounted Kinect and converted to an EDT representation which is then used for planning (top right). The robot is modeled as a set of spheres (red) positioned according to the joint angles and forward kinematics. The initial path goes through the obstacle (bottom left), the planned trajectory avoids it (bottom right). The companion video shows how the complete experiment including execution of the planned path is performed on the real robot.

computation which can be used with any gradient-based OMP variant (e.g. quasi-Newton, conjugate gradient, or the stochastic Hamilton Monte Carlo methods). To close the loop in the experiment, we have chosen a standard optimizer to plan a trajectory that avoids an obstacle in a tabletop scene. Our system is able to handle static scenes in a manipulation context where model acquisition (sensor sweeps, etc.), model post processing and motion planning may take about a second each.

The remainder of the paper is structured as follows. After a discussion of related work (II), we introduce optimization-based motion planning as a general method (III), briefly explain key GPGPU (CUDA) concepts (IV), show how to transform KinectFusion models into a representation suitable for planning (V,VI), how to compute the motion planning objective functions and gradients from this (VII,VIII), present our experimental results (IX), and close with conclusions and future work (X).

R. Wagner and B. Bäuml are with DLR Institute of Robotics and Mechatronics, 82234 Wessling, Germany. U. Frese is with German Research Center for Artificial Intelligence (DFKI) 28359 Bremen, Germany. R. Wagner and U. Frese are also with University of Bremen, 28359 Bremen, Germany. Contact {rene.wagner,berthold.baeuml}@dlr.de, udo.frese@dfki.de.

II. RELATED WORK

A. 3D Modeling

The introduction of the Microsoft Kinect sensor has had a fairly disruptive effect on the robotics community. Never before has a sensor provided depth data at the same frame rate (30Hz), resolution (640x480 depth values per frame), density (virtually no “holes”), accuracy (depth error $< 3\text{cm}$ at 2m distance), with robustness against low texture environments (due to active stereo replacing one stereo camera with an IR pattern projector), all in a fairly compact, inexpensive package.

The first algorithm to fully leverage the Kinect depth data was KinectFusion [13]. The unique properties of KinectFusion are that it uses depth data to both track the sensor pose and build the environment model (map) rather than relying on RGB feature based SLAM and mapping with known poses as earlier approaches [8]. It is thus less prone to problems due to lighting or lack of texture, but needs environment geometry that constrains the sensor pose (e.g. it is problematic if only a single, entirely flat surface is in view). Another unique property is that KinectFusion manages to process the vast amount of all depth data at frame rate thanks to a highly parallelized GPGPU implementation (NVIDIA CUDA).

As far as the generated model (map) is concerned, like earlier methods, KinectFusion uses a voxel grid (although at a very high resolution – with a voxel side length typically $\leq 6\text{mm}$), but does not just maintain occupancy information [19]. Instead, it stores the signed distance to the nearest surface in each voxel, thus generating a sub-voxel surface model, and facilitating averaging over distance values as subsequent depth measurements are fused with the model. Over time, this eliminates sensor noise and if the sensor is moved also depth discretization errors. The result is an unprecedentedly dense, smooth and accurate 3D model that is instantly available due to operation at frame rate.

B. Motion Planning

For motion planning in high-dimensional spaces the three most popular classes of algorithms are:

- sampling-based motion planning (e.g., PRM [12] and RRT [11]) which first searches for a collision-free path by random sampling and then optimizes this path for a given criterion (e.g., smoothness),
- search-based motion planning [5] which formulates planning as a search problem in a graph of motion primitives resulting in a smooth collision-free path and
- optimization-based motion planning (OMP) [16, 17, 20], which computes an optimal path by minimizing a functional objective comprised of an obstacle avoidance and an optimality criterion (e.g., smoothness or energy optimality) term.

For precisely controlled and dynamically performant humanoid robots like DLR’s Justin [1] OMP is very attractive as it allows for direct planning of precise reaching motions close to obstacles as needed in dexterous manipulation tasks (smoothness) or for the planning of fast whole-body motions

(dynamical optimality), e.g., when catching a ball (see [1] for Justin in action). In addition, two major drawbacks of OMP have been removed recently by algorithms for efficient computation of the objective and its gradient and the alleviation of the problem of local minima by using modern stochastic optimization methods (e.g., Hamilton Monte Carlo [20] or stochastic path integrals [10]).

To enable a robot to operate really autonomously in unknown and cluttered environments, the sensor-model-planning-action loop has to be efficiently closed. That esp. implies that the planning algorithms have to be able to work on models acquired from sensor data and that the model acquisition and the planning are fast. It is only recently that robustly working implementations have been shown using a stack of processing modules [4, 9]:

- computing point clouds from 3D-sensors (e.g., Kinect or laser scanner),
- outlier-filtering,
- integration in a probabilistic voxel-map (about 2cm resolution) representing obstacles, free- and unknown space (Octomap [19]) and
- sample-based planning for one arm (up to 12 DOF).

This differs from our approach in important aspects:

- We perform almost all processing directly on the GPU,
- raw data is directly integrated in a high-resolution (2mm) model (which is also used in the sensor tracking step) and
- an optimization-based planner computes paths for a full humanoid with 19 DOF.

Recently, GPU-implementations for sample-based planners or the important part of obstacle-avoidance checking have been presented [15]. However, none of them compute an overlap volume or its gradient as needed for an OMP.

III. OPTIMIZATION-BASED PLANNING

In optimization-based planning, the problem of finding a collision-free and optimal path from a given start to a given goal configuration is formulated as a minimization problem of a functional of the path. Here, we give an overview of the continuous formulation of the method following [20], in VII we present the discretized version as we have implemented it.

The objective functional $U(q)$ of a path $q(t)$ is comprised of a collision term $U_c(q)$, which penalizes collisions of the robot with obstacles, and a smoothness term $U_s(q)$, which penalizes non-smooth paths, scaled with a weighting factor λ . The equations adapted from [20] read as follows:

$$U(q) = U_c(q) + \lambda U_s(q) \quad (1)$$

$$U_s(q) = \frac{1}{2} \int_0^1 \left\| \frac{d}{dt} q(t) \right\|^2 dt \quad (2)$$

$$U_c(q) = \int_0^1 \int_B c(x(q(t), u)) \left\| \frac{d}{dt} x(q(t), u) \right\| du dt \quad (3)$$

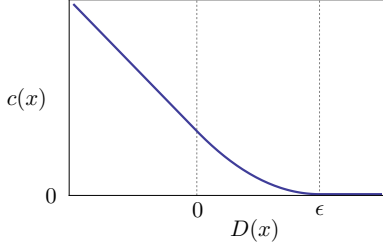


Fig. 2. The penetration depth function $c(x)$ is essentially a clipped version of the signed distance function $D(x)$ adding an ϵ -transition range. $D(x)$ measures for each point x the distance to the nearest obstacle boundary (negative values inside obstacles, positive outside). The ϵ -range basically leads to smooth gradients at the border of occupied and free space.

$$c(x) = \begin{cases} -D(x) + \frac{1}{2}\epsilon & \text{if } D(x) \leq 0 \\ \frac{1}{2\epsilon} (D(x) - \epsilon)^2 & \text{if } 0 < D(x) \leq \epsilon \\ 0 & \text{otherwise,} \end{cases} \quad (4)$$

where u runs over the robot's volume B . Here, $q(t)$ is an arbitrary parameterization of the robot's motion, e.g., the joint angles, as a function of an arbitrary path variable t (in the rest of the paper, we use the intuition of t being a time) and $x(q(t), u)$ is the cartesian coordinate of a point of the robot, statically identified by u , e.g. link number and coordinate in the link fixed coordinate system.

Intuitively, $U_c(q)$ measures the overlap of the swept volume of the robot (i.e., the volume the robot sweeps over during moving along the path) and the obstacles. Hence, to minimize $U_c(q)$ the optimizer tries to push the robot out of obstacles. To make this pushing effect more robust and, e.g., even work when a link of the robot is completely inside an obstacle, the overlap volume is weighted with a penetration-depth field $c(x)$ penalizing deeper penetration more (see Fig. 2 and (4)). The key to making the objective invariant to parameterization is the multiplication with the norm of the Cartesian velocity: $\|\frac{d}{dt}x(q(t), u)\| dt$ measures an arc length along the trajectory.

The smoothness term $U_s(q)$ is the square of the joint angle velocities and minimizing it tries to straighten out or smoothing the path.

Fast converging optimization methods, like quasi-Newton or conjugate gradients, can take advantage of also providing the gradient of the objective function. In the continuous formulation the functional gradient of (1) has to be calculated and reads as [20]

$$\frac{\delta U}{\delta q} = \frac{\partial v}{\partial q} - \frac{d}{dt} \frac{\partial v}{\partial \dot{q}} \quad (5)$$

where v refers to all terms under the time integral. For the collision term this results in [20]

$$\frac{\delta U_c}{\delta q} = \int_B J^T \|\dot{x}\| [(I - \hat{x}\hat{x}^T) \nabla c - c\kappa] du \quad (6)$$

$$\kappa = \|\dot{x}\|^{-2} (I - \hat{x}\hat{x}^T) \ddot{x} \quad (7)$$

$$\nabla c(x) = \begin{cases} -\nabla D(x) & \text{if } D(x) \leq 0 \\ \frac{1}{2} (D(x) - \epsilon) \nabla D(x) & \text{if } 0 < D(x) \leq \epsilon \\ 0 & \text{otherwise,} \end{cases} \quad (8)$$

where $\dot{x} = \frac{d}{dt}x(q(t), u)$ and $\hat{x} = \dot{x} / \|\dot{x}\|$ denotes the unit vector in the velocity direction and κ is the curvature of the path.

Looking at (6) it is important to note that, given the spatial gradient of the penetration-depth field $\nabla c(x)$ and the Jacobian J of the forward kinematics for each point x , the computation of the functional gradient $\frac{\delta U_c}{\delta q}$ is only slightly more expensive than computing $U_c(q)$ itself! This fact is exploited to efficiently compute the discretized version of (6) in section VII.

IV. GPGPU PROGRAMMING USING CUDA IN A NUTSHELL

NVIDIA CUDA is essentially a subset of C++ (template support is very limited) with some extensions that allow for GPU and CPU code to co-exist in the same source file. GPU and CPU code are passed to separate compilers. A runtime API allows for GPU code to be loaded to and executed on the GPU as well as copying of memory to and from the GPU. For the purposes of this paper it will suffice to understand the execution model as this is how the parallel structure of a problem is encoded.

CUDA code that solves a certain task is wrapped in a so-called *compute kernel*. The same kernel is executed in the form of multiple *threads* in parallel where each thread typically performs the same instruction on different data items. The GPU contains several streaming multiprocessors (SMs) each of which operates independently from the others and which in turn contains a number of compute *cores*, e.g. the GTX 680 contains eight SMs consisting of 192 cores each [14]. The exact mapping from logical threads to physical cores is hidden behind an abstraction that groups threads into logical blocks. When launching a kernel, one needs to specify how many blocks and how many threads within each block are to be executed. A single block cannot span multiple SMs and must not exceed available resources (registers, shared memory, etc.).

Conveniently, with the present generation of NVIDIA GPUs, both the number of threads and the number of blocks to be executed can be specified in three dimensions each. When a kernel is executed each thread can programmatically access the indices of its position in the block of threads and in the grid of all blocks. Thus, parallelization in CUDA essentially boils down to mapping problem-specific dimensions to CUDA block and grid dimensions.

V. FROM KINECTFUSION TO MOTION PLANNING

At the core of the KinectFusion algorithm [13] is a variant of the iterative-closest-points (ICP) algorithm [2] which, until convergence, alternately estimates the transformation between two sensor poses and uses this new estimate to projectively determine the data association of point clouds obtained from these poses.

The reference point cloud is computed from a raycast of the previous model (typically a 512^3 voxel grid) as viewed from the previous sensor pose, the second point cloud is computed from the current depth image obtained from the

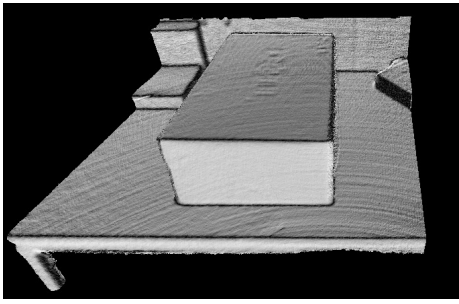


Fig. 3. 3D model of the tabletop scene generated with our KinectFusion reimplementation and rendered using a simple lighting model based on the raw surface normals (i.e. no smoothing/Phong shading) on the GPU. Note that horizontally the model is smoother than vertically since the robot’s head only panned but did not tilt in this experiment so that the depth discretization cannot be “averaged away”. This does not occur with hand-held operation due to the inherent shakiness of human motion.

sensor. The current depth image is then registered in the model according to the new sensor pose obtained from ICP to yield the new model and this is repeated every time a new depth image is received from the sensor. KinectFusion operates a frame rate (30Hz) by exploiting the parallel structure of the problem (pixels in the depth image, voxels in the model grid are largely independent) and offloading virtually all computation onto the GPU.

We use a custom KinectFusion re-implementation which achieves the same performance in terms of computing time and the (visually compared) quality of generated models. There are many tradeoffs and tunable parameters involved, so our generated models are not necessarily identical. However, we believe the only significant modification in our implementation is that we use the familiar radial distortion model $r' = 1/(1 + \kappa r^2)$ while the original implementation neglects distortion. In our experience an accurate sensor model and calibration are crucial. We use the calibration method from our earlier work [18]. The model generated from our tabletop evaluation scene is depicted in Fig. 3.

To use models generated by KinectFusion for planning, it is important to understand its truncated signed distance function (TSDF) model representation. For every voxel p in the voxel grid, it stores a weight $W(p)$ and the truncated signed distance $F(p)$ to the nearest surface. Voxels with $W(p) = 0$ have never been touched, corresponding to unknown space. With $D(p)$ being the signed Euclidean distance to the nearest surface, the relevant intuition behind $F(p)$ can be captured as follows:

$$F(p) = \begin{cases} 1 & \text{if } D(p) \geq \mu \\ \frac{D(p)}{\mu} & \text{if } -\mu \leq D(p) < \mu \\ \text{undefined} & \text{otherwise} \end{cases} \quad (9)$$

Thus, within $\pm\mu$ around surfaces the TSDF directly corresponds to the Euclidean distance needed for planning. The truncation unfortunately prevents a direct lookup of distances needed for planning but this truncation allows KinectFusion to handle multiple surfaces so that, e.g., objects thicker than 2μ can be viewed from the front and back without surface

measurements cancelling out each other.

However, conversion to obstacle information is easy. Conservatively treating unknown space as an obstacle, the occupancy of a voxel can be determined as follows:

$$Occupancy(p) = \begin{cases} FreeSpace & \text{if } W(p) > 0 \wedge F(p) > 0 \\ Obstacle & \text{otherwise} \end{cases} \quad (10)$$

Note how this automatically fills the insides of objects. This binary information can then be used to recover the Euclidean distance for every voxel as shown in the following section.

VI. COMPUTING THE EUCLIDEAN DISTANCE TRANSFORM (EDT) AND ITS GRADIENT ON THE GPU

The Euclidean Distance Transform (EDT) of a function f is defined as [6]

$$D_f(p) = \min_q (\|p - q\| + f(q)) \quad (11)$$

We use f to encode obstacle and free space information, i.e. to compute the distance from free space to the nearest obstacle we set

$$f(q) = \begin{cases} \infty & \text{if } q \text{ is } FreeSpace \\ 0 & \text{if } q \text{ is } Obstacle \end{cases} \quad (12)$$

which ensures that $(\|p - q\| + f(q))$ is minimal at the nearest obstacle.

A naive implementation of (11) would be $O(n^2)$ where n is the number of voxels. The key insight towards a fast implementation is to compute the squared EDT and exploit that in the 2D case [6]

$$\begin{aligned} D_f(x, y) &= \min_{x', y'} ((x - x')^2 + (y - y')^2 + f(x', y')) \\ &= \min_{x'} ((x - x')^2 + \min_{y'} ((y - y')^2 + f(x', y'))) \\ &= \min_{x'} ((x - x')^2 + D_{f|_{x'}}(y)). \end{aligned} \quad (13)$$

This extends to arbitrary dimensions and essentially boils down to the application of the Pythagorean theorem. Thus, we can compute the squared EDT of the 3D grid by successively computing the squared EDT along a single dimension at a time.

Like CHOMP [20], we use the algorithm presented in [6] to do the latter – except we run it on the GPU. It has a time complexity of $O(n)$ where n is the maximum extent along a single dimension. It first analyzes the algebraic structure computing the lower envelope of n parabolas (cf. (11)) and from this determines the squared EDT in a second pass. The linear time complexity comes at a price: space complexity – the lower envelope computation needs $O(n)$ memory.

Thus, while the algorithm from [6] can be transcribed almost verbatimly into C++, its execution on the GPU is not straight-forward: The algorithm works by looping over a complete single column of the volume, later values potentially depend on all previous values. The only way to parallelize this is to have $n \times n$ parallel threads process one column each. Each thread then needs $O(n)$ memory easily exceeding the available GPU memory.

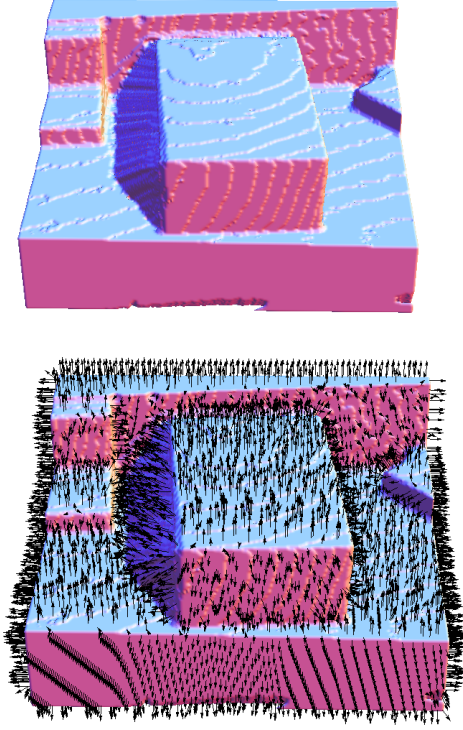


Fig. 4. Contour plot of generated EDT (top) and its gradient (bottom).

To cope with this, we first downsample the 512^3 TSDF grid to a 256^3 EDT grid. Additionally, note that the memory actually accessed at any time is limited by the number of runnable threads which in turn are limited by the number and size of streaming multiprocessors on the GPU.¹ We take advantage of this by splitting the EDT computation into multiple smaller kernel launches – each just large enough to keep the GPU busy. While this direct implementation of [6] on the GPU works, we plan to investigate other exact EDT algorithms more tailored towards the GPU [3] (but implementation-wise also much more involved as it needs to merge partial results in a tree-style fashion) in future work.

The squared EDT computation is executed once for the distance of free space voxels from the nearest obstacle voxels and once the other way around. The results are merged and the square root is taken in a post-processing step.

The computation of the EDT gradient is straight forward: We use a $3 \times 3 \times 3$ Sobel filter and appropriate scaling to approximate the true gradient. Despite the downsampling and the small filter size the resulting gradient is surprisingly smooth (Fig. 4).

VII. COMPUTING THE OBSTACLE-AVOIDANCE OBJECTIVE FUNCTION AND GRADIENT

We use a simple robot model comprised of a number of spheres for each link. The forward kinematics is executed

¹With the GTX 680, according to the `cudaGetDeviceCount` API call the maximum number of runnable threads is $8 \cdot 2048 = \text{multiProcessorCount} \cdot \text{maxThreadsPerMultiProcessor}$.

on the CPU and we transmit the model to the GPU as follows. Each sphere is represented by its center $x_{l,i}^{B_l}$ in the coordinate system of its link B_l and its radius $r_{l,i}$. The pose of each link at each time step is transmitted as the set of $SE(3)$ transformations $F_{B_l(t_j)}^0$ with corresponding partial derivatives $\frac{\partial F_{B_l(t_j)}^0}{\partial q_k(t_j)}$ of the frames with respect to the joint angles $q_k(t_j)$ where k identifies each joint.

We discretize the objective and gradient formulae from III in the time domain according to [20] as follows. The objective U at time t_j is the sum of the penetration depths over all links l and spheres i weighted with their respective velocities:

$$U(t_j) = \sum_l \sum_i c(x_{l,i}(t_j)) \|\dot{x}_{l,i}(t_j)\| \quad (14)$$

where the sphere centers are transformed to global coordinates via

$$x_{l,i}(t_j) = F_{B_l(t_j)}^0 \cdot x_{l,i}^{B_l} \quad (15)$$

and the corresponding velocities and accelerations are computed through finite differences:

$$\dot{x}_{l,i}(t_j) = \frac{x_{l,i}(t_j) - x_{l,i}(t_{j-1})}{\Delta t} \quad (16)$$

$$\ddot{x}_{l,i}(t_j) = \frac{\dot{x}_{l,i}(t_j) - \dot{x}_{l,i}(t_{j-1})}{\Delta t}. \quad (17)$$

The Jacobians are computed from the frame derivatives taking the position of each sphere within its link (the added “lever arm”) into account:

$$J_{l,i,k}(t_j) = \frac{\partial F_{B_l(t_j)}^0}{\partial q_k(t_j)} \cdot x_{l,i}^{B_l}. \quad (18)$$

With $\hat{x} = \frac{x}{\|x\|}$, the time-discretized partial derivatives of the objective with respect to each joint angle q_k are then:

$$\begin{aligned} \frac{\partial U(t_j)}{\partial q_k(t_j)} = & \sum_l \sum_i \left(J_{l,i,k}^T(t_j) \|\dot{x}_{l,i}(t_j)\| \right. \\ & \cdot \left((I - \hat{x}_{l,i}(t_j) \hat{x}_{l,i}^T(t_j)) \nabla c(x_{l,i}(t_j)) \right. \\ & \left. \left. - \frac{c(x_{l,i}(t_j))}{\|\dot{x}_{l,i}(t_j)\|^2} (I - \hat{x}_{l,i}(t_j) \hat{x}_{l,i}^T(t_j)) \ddot{x}_{l,i}(t_j) \right) \right). \quad (19) \end{aligned}$$

To be numerically more stable we rewrote (19) using $(I - \hat{x} \hat{x}^T)b = b - \frac{(\hat{x} \cdot b) \hat{x}}{\|\hat{x}\|^2}$ to yield

$$\begin{aligned} \frac{\partial U(t_j)}{\partial q_k(t_j)} = & \sum_l \sum_i \left(J_{l,i,k}^T(t_j) \cdot \right. \\ & \cdot \left(\|\dot{x}_{l,i}(t_j)\| \nabla c(x_{l,i}(t_j)) - \frac{(\dot{x}_{l,i}(t_j) \cdot \nabla c(x_{l,i}(t_j))) \dot{x}_{l,i}(t_j)}{\|\dot{x}_{l,i}(t_j)\|} \right. \\ & \left. \left. - \frac{c(x_{l,i}(t_j))}{\|\dot{x}_{l,i}(t_j)\|} \left(\ddot{x}_{l,i}(t_j) - \frac{(\dot{x}_{l,i}(t_j) \cdot \ddot{x}_{l,i}(t_j)) \dot{x}_{l,i}(t_j)}{\|\dot{x}_{l,i}(t_j)\|^2} \right) \right) \right). \quad (20) \end{aligned}$$

We can now instantiate (14) and (20) to obtain the obstacle avoidance objective and gradient respectively by setting

$$D(x) := EDT(x_{l,i}(t_j)) - r_{l,i} \quad (21)$$

and

$$\nabla D(x) := \nabla EDT(x_{l,i}(t_j)). \quad (22)$$

Parallelization for execution on the GPU is now straightforward: For each combination of l, i and t_j for the objective and l, i, t_j and q_k for the gradient, we can compute the respective summand in parallel. We map the l and i s to the dimensions of each CUDA block and the j and k s to the dimensions of the CUDA grid. The summation is computed by means of a tree-style parallel reduce operation [7]. We have compared the results of our GPU implementation with an independently developed Mathematica implementation and they turn out to be virtually identical when both implementations operate on same EDT and EDT gradient despite the fact that the GPU uses single-precision floating point operations. We primarily attribute this to the fact that the tree-based summation is numerically more stable than a (naive) sequential summation.

VIII. COMPUTING THE SELF-COLLISION-AVOIDANCE OBJECTIVE FUNCTION AND GRADIENT

We essentially need to determine the depth of the self-penetration of the robot. To compute this, we test, for each sphere in the robot model, this one moving sphere against the static rest of the robot. Since some spheres always cause self-collisions we extend the above interface with a blacklist specifying which pairs of (l, i) and (l', i') are to be ignored (zero objective and gradient). Note that this is an approximation since in reality the penetration depth depends on how spheres move mutually relative to one another but the approximation is conservative, i.e. the approximated penetration depth is always larger.

As the structure is the same as above, we can once again instantiate (14) and (20) to yield the obstacle avoidance objective and gradient respectively this time by setting

$$D(x) := \|x_{l,i}(t_j) - x_{l',i'}(t_j)\| - (r_{l,i} + r_{l',i'}) \quad (23)$$

and

$$\nabla D(x) := \frac{x_{l,i}(t_j) - x_{l',i'}(t_j)}{\|x_{l,i}(t_j) - x_{l',i'}(t_j)\|} \quad (24)$$

and performing the summation over two additional indices l' and i' .

Note that this time the summation needs to be computed over four indices in total (l, i, l', i') exceeding the maximum number of dimensions in a CUDA block. Thus, we need to map the l' s to one of the CUDA grid dimensions and use a two-stage reduction first computing the sum over a block and then over the result of all blocks with the same l' . Like above, we have validated the GPU-computed results against a Mathematica re-implementation and, remarkably, they turn out to be identical up to 10^{-6} .

IX. EXPERIMENTAL VALIDATION

To join all previous components into a functional prototype we have implemented the forward kinematics in Mathematica and use the standard Mathematica optimization solver which calls our CUDA implementation to compute the objective functions and gradients. The model generation and planning steps are currently invoked manually. For the experimental setup, we use the setup depicted in Fig. 1. We use real data

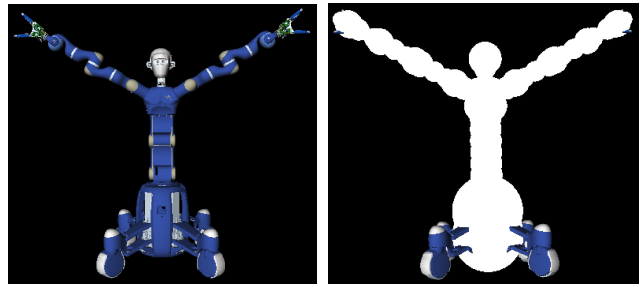


Fig. 5. CAD model of Agile Justin (left) and overlaid sphere model for self-collision avoidance (right). The legs are currently not modeled. The hands are modeled in their closed position. Fingers are currently not modeled individually.

TABLE I
PARAMETERS INFLUENCING THE PROBLEM SIZE.²

Number of Joints	19
Number of Links	16
Number of Spheres per Link	4
Number of Time Steps	50

acquired from the head-mounted Kinect to generate the 3D environment model (Fig. 3). The KinectFusion grid size is 512^3 with a voxel side length of 2mm, the EDT grid consists of 256^3 voxels with a side length of 4mm each.

The start and goal joints configurations with the TCP to the left and to the right of an obstacle on a tabletop space have been recorded from encoder readings. It should be noted that although the goal is formulated for the TCP, a path for the full 19 DOF of Agile Justin is to be planned, not just for the arm configuration. The robot model consists of 16 links and 4 spheres per link and fully encloses the robot (Fig. 5). Thanks to Agile Justin's round exterior shapes the model is also very close to the true robot surface.

Tab. I illustrates the overall problem size and shows why GPGPU parallelization is beneficial. In terms of processing hardware, we use an Intel Xeon E5-2665 @ 2.40GHz and an NVIDIA GTX 680 GPU with 2GB GPU memory (8 streaming multiprocessors, 192 CUDA cores each).

The resulting trajectory after 100 calls of the GPU-based objective/gradient computations is illustrated in Fig. 1 and its execution is shown in the companion video.

The relevant computing times are given in Tab. II. The KinectFusion step is slightly slower than reported in the original paper since we use a finer step size in the raycaster. As in the original implementation, the KinectFusion time is not constant primarily because the raycasting step depends on the model and view angle. Computation of the EDT takes orders of magnitude longer than the other steps since the algorithm does not parallelize well as discussed in section VI and requires unsystematic (data dependent) reads and writes to large amounts of temporary memory. The EDT gradient computation is again fast and constant time. The obstacle objective is constant time if the overlap of the robot model

²The number of links is smaller since due to rotational symmetry some joints have no effect on the shape of the current sphere model.

TABLE II
COMPUTING TIMES OF INDIVIDUAL STEPS.

Step	Time [ms]
KinectFusion (per Frame)	27-31
EDT (once)	1150
EDT Gradient (once)	32
Obstacle and Self-Collision Objective & Gradient (per evaluation)	8

with the EDT grid is constant (the area outside the grid is assumed to be free space). The self-collision objective is always constant time.

It should be noted that in our current implementation the memory access pattern of both objective computations is not ideal (i.e. not coalesced). This prevents simultaneous fetching of larger memory blocks and typically has a significant impact on memory throughput in any CUDA application.

X. CONCLUSIONS AND FUTURE WORK

We have presented a complete, functional proof-of-concept GPGPU implementation of a 3D environment modeling and motion planning system and evaluated it on real data on a 19-DOF humanoid robot. Most notably, the environment model used for model generation contains all information required for planning and is never transferred to the CPU – virtually all computation up to the point where the planning objectives and gradients have been determined happens on the GPU. The computing time of the overall system is dominated by the EDT computation (≈ 1 s). This is fast for planning on fine resolution models as needed for dexterous manipulation.

In future work, we intend to further optimize the obstacle and gradient computation as noted above and investigate moving the complete optimizer onto the GPU. We will also increase the EDT model resolution to 2mm by upgrading to a 4GB version of the GPU card.

ACKNOWLEDGMENTS

This work was partly supported under DFG grant SFB/TR 8 Spatial Cognition.

REFERENCES

- [1] B. Bäuml, F. Schmidt, et al. Catching Flying Balls and Preparing Coffee: Humanoid Rollin’Justin Performs Dynamic and Sensitive tasks. In *Proc. IEEE Int. Conf. on Robotics and Automation*, 2011.
- [2] P. J. Besl and N. McKay. A method for registration of 3-D shapes. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 14(2):239 – 256, 1992.
- [3] T.-T. Cao, K. Tang, A. Mohamed, and T.-S. Tan. Parallel banding algorithm to compute exact distance transform with the gpu. In *Proc. ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, 2010.
- [4] S. Chitta, E. G. Jones, M. Ciocarlie, and K. Hsiao. Perception, planning, and execution for mobile manipulation in unstructured environments. *IEEE Robotics and Automation Magazine*, 19(2):58–71, 2012.
- [5] B. J. Cohen, S. Chitta, and M. Likhachev. Search-based planning for manipulation with motion primitives. In *Proc. IEEE Int. Conf. on Robotics and Automation*, 2010.
- [6] P. F. Felzenszwalb and D. P. Huttenlocher. Distance transforms of sampled functions. Technical Report TR2004-1963, Cornell University, 2004.
- [7] M. Harris, S. Sengupta, and J. D. Owens. Parallel prefix sum (scan) with cuda. In H. Nguyen, editor, *GPU Gems 3*, chapter 39, pages 851 – 876. Addison Wesley, 2007.
- [8] P. Henry, M. Krainin, et al. RGB-D mapping: Using depth cameras for dense 3D modeling of indoor environments. In *Proc. Int. Symposium on Experimental Robotics*, 2010.
- [9] A. Hornung, M. Phillips, et al. Navigation in three-dimensional cluttered environments for mobile manipulation. In *Proc. IEEE Int. Conf. on Robotics and Automation*, 2012.
- [10] M. Kalakrishnan, S. Chitta, et al. Stomp: Stochastic trajectory optimization for motion planning. In *Proc. IEEE Int. Conf. on Robotics and Automation*, 2011.
- [11] S. Karaman and E. Frazzoli. Sampling-based algorithms for optimal motion planning. *Int. Journal of Robotics Research*, 30(7):846–894, 2011.
- [12] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, 2006.
- [13] R. A. Newcombe, S. Izadi, et al. KinectFusion: Real-time dense surface mapping and tracking. In *IEEE Int. Symposium on Mixed and Augmented Reality*, 2011.
- [14] NVIDIA. NVIDIA GeForce GTX 680, 2012. Whitepaper.
- [15] J. Pan and D. Manocha. GPU-based parallel collision detection for fast motion planning. *The Int. Journal of Robotics Research*, 31(2):187–200, 2012.
- [16] S. Quinlan and O. Khatib. Elastic bands: Connecting path planning and control. In *Proc. IEEE Int. Conf. on Robotics and Automation*, 1993.
- [17] N. Ratliff, M. Zucker, J. A. Bagnell, and S. Srinivasa. CHOMP: Gradient optimization techniques for efficient motion planning. In *IEEE Int. Conf. on Robotics and Automation*, 2009.
- [18] R. Wagner, O. Birbach, and U. Frese. Rapid development of manifold-based graph optimization for multi-sensor calibration and SLAM. In *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, 2011.
- [19] K. M. Wurm, A. Hornung, et al. OctoMap: A probabilistic, flexible, and compact 3D map representation for robotic systems. In *Proc. ICRA 2010 Workshop on Best Practice in 3D Perception and Modeling for Mobile Manipulation*, 2010.
- [20] M. Zucker, N. Ratliff, et al. CHOMP: Covariant hamiltonian optimization for motion planning. *Submitted to Int. Journal of Robotics Research*, 2012. Preprint: www.cs.cmu.edu/~sidh/preprints/CHOMP12-ijrr.pdf.